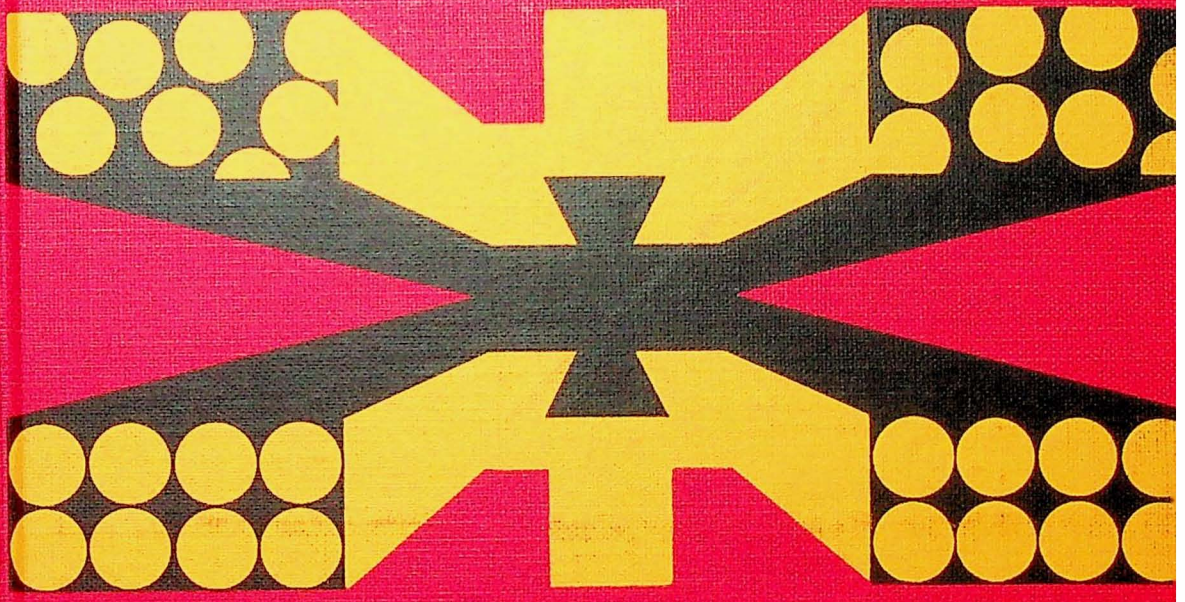
PROGRAMMING THE IBM SYSTEM/360



ROBERT E. STEINHART
SEYMOUR V. POLLACK

PROGRAMMING THE IBM SYSTEM/360

Robert F. Steinhart
IBM Corporation

Seymour V. Pollack
Washington University

HOLT, RINEHART AND WINSTON
NEW YORK CHICAGO SAN FRANCISCO ATLANTA
DALLAS MONTREAL TORONTO LONDON SYDNEY

Copyright © 1970 by Holt, Rinehart and Winston, Inc.
All Rights Reserved

Library of Congress Catalog Card Number 70-107437

SBN: 03-079175-8

Printed in the United States of America

0 1 2 3 22 9 8 7 6 5 4 3 2 1

Dedicated to all those who have encouraged us, especially our wives, and to our children whose generation, hopefully, will greatly benefit from the computer as an agent of social good.

Preface

To a certain extent, our understanding of the impact of computers has been following roughly the same developmental pattern associated with the automobile. When the latter device was introduced, relatively few people thought about its effects beyond the context of an instrument for transporting people and goods further, faster, and potentially cheaper. The fundamental changes in our social structure, cities, family patterns, and even in our sexual attitudes, all directly attributable to the automobile, were largely unanticipated fifty years ago. In the late 1940's and even into the early '50's the same type of thinking generally prevailed with respect to computers. Since these were considered to be ultra-fast calculating devices suitable for solving large systems of complex mathematical problems, the need for computers and their scope of applications have been consistently underestimated.

Our predictive accuracy has been particularly unimpressive with regard to the time scale over which the growth of computer-related technology was expected to take place. The onrush of developments in hardware and software, coupled with an ever growing array of new applications, forces a continual revision of such predictions. In fact, it would be difficult to refute the notion that this rate of development is unique in the history of science. Furthermore, the growth and expansion processes have been chaotic, adding to the difficulty of early extrapolation.

Consequently, it was necessary for some time to pass for a general trend to become clear. As has happened in so many other areas, the availability of a "past" has now made it possible for analysts and commentators to superimpose a systematic developmental history which puts things in better order (with a little pressure on the time sequence here and there) so that prediction is facilitated. In the case of computer technology the rapid

growth rate shortened the amount of time required for such background to pile up. Thus, although it is only a scant 20 years since digital computers were generally introduced, a large number of people are already thinking of them in terms of their broad influence on society rather than merely as tools.

A series of landmarks which serves as a particularly convenient vehicle for constructing a cohesive history of the field's development is the concept of a *generation* of computers. It turns out that certain significant advances in programming theory, systems design concepts, and algorithmic techniques can be linked directly to the introduction of certain design features in computer systems. Consequently, the beginning of a new generation has been associated with the introduction of those types of computers which provide these features for the first time. Of course, the issue is never so clear-cut. First of all, there is no universal agreement as to which combinations of features constitute a generation. Secondly, most of the features generally thought of in this context are not mutually dependent, so that it is not unusual for different machines to present different combinations of the features. Thus, a computer assigned to one generation often possesses some attributes associated with an earlier or a later one.

Despite these inconsistencies, it is generally accepted that we are in the third generation of computer development. Furthermore, it is recognized that this generation does not represent merely an extension of prior technology, as was the case with the second generation. The introduction of such features as operating systems, multiprogramming, large capacity modular memories, extended character sets, and elaborate interrupt capabilities, along with fundamental advances in circuitry and programming languages, are simply too immense to be considered within that frame of reference. Thus, it is fairly easy to predict that the time required to absorb, exploit, and outgrow these innovations will substantially lengthen the span of this generation.

Because of its widespread acceptance among users, the IBM System/360 series of computers is generally considered to be the group that is most characteristic of the third generation. As such, these machines serve as the vehicle for much of the information processing in this country and elsewhere, as well as considerable programming effort. Hence, a programmer with this orientation can be said to be operating within the mainstream of third generation technology.

This book is intended to introduce the wide range of features available to the System/360 programmer. It is not oriented towards beginners but to individuals who have some familiarity with computing and programming concepts, gained in the perspective of earlier equipment or other third generation systems. Consequently, no attempt is made to explain in detail the principles behind computer operation, software design, or program development. Instead, the book emphasizes the presentation of the ex-

tensive array of programming capabilities available through the System/360 by means of various programming languages, such as Assembler Language, FORTRAN IV, COBOL, PL/I, and ALGOL.

It is important to say something about the depth of coverage. Each of the languages mentioned above is an extensive logical structure and represents a major software effort. Consequently, its presentation and exploration in this book, though quite broad, is not exhaustive. Coverage is sufficiently detailed so that the reader, after working with a language in this book, will understand the overall structure of the language and will have sufficient fluency in it to make effective use of all but its most advanced features. Those programmers seeking multilingual capability will find this book extremely useful in selecting the most suitable language for a given problem, based on the effectiveness with which the 360's capabilities are utilized in that language for that problem, as well as the nature of the problem itself.

To provide this level of familiarity, this book is organized as follows:

1. The first three chapters present basic material on the organizational and operational characteristics of the System/360 (Model 25 and larger). The very close operational relationship between hardware and software is established and is considered in conjunction with basic internal data storage and handling procedures to introduce the reader to Assembler Language.
2. The next eight chapters are devoted to a detailed presentation of 360 Assembler Language. During this section the student is guided through the various types of symbolic instructions, allowing him to build a working capability in this language. This material, in combination with the background presented earlier, permits the student to evaluate the efficiency of alternative instruction sequences for various procedures and serves as excellent preparation for eventual entry into systems work. The techniques presented here are reinforced with numerous illustrative examples and problems.
3. The Assembler Language chapters are followed by a section on each of the three common high level languages (FORTRAN IV, COBOL, and PL/I). Each of these languages is presented in detail so that the programmer, if he wishes, can learn to use any or all of them routinely. Problems are selected to emphasize the strengths and weaknesses of each language in various contexts. Special attention is paid to the possible use of PL/I as a *universal* language which includes many of the features usually associated either with FORTRAN or COBOL.
4. ALGOL, while not as widely used as the other high level languages, is very powerful and provides useful insights into the structure of artificial languages, often serving as a model for many of them. Since

it is available on many System/360 configurations, it is described in an appendix.

5. Although the introductory chapters describe the basic mechanism for communication between the 360 programmer (regardless of the language he uses) and the supervisory software, some readers may wish to explore this further. For this purpose, various basic features of the Job Control Language are presented in a separate appendix. The reader using that information will gain a more detailed understanding of the capabilities and versatility of 360 operating systems, allowing him to make use of his own particular computing facility at a higher level of sophistication.

Because of this type of organization, this book can be used by programmers at all levels who wish to write programs for the System/360. The extensive use of illustrative examples and the variety of problems and exercises make it a suitable programming text for students who are familiar with basic concepts. The introductory material in the initial chapter has been found particularly useful for management and supervisory personnel with minimal technical orientation who wish to become familiar with third generation technology.

April 1970

ROBERT F. STEINHART
SEYMOUR V. POLLACK

Contents

Preface v

Chapter 1

Introduction to System/360 Organization 1

1.1 System Organization 1
1.2 Hardware 11
1.3 Software 30
Problems 40
References 41

Chapter 2

System/360 Data Handling 44

2.1 The System/360 Alphabet 44
2.2 Organization of Memory 47
2.3 Internal Representations of Numbers 53
Problems 62

Chapter 3

Basic Instruction Handling 64

3.1 Basic Instruction Formats 64
3.2 Basic Instruction Types 70
3.3 Processing of Instruction Sequences 78
3.4 Machine Indicators 83
Problems 91
References 92

Chapter 4
Introduction to Assembler Language **93**

4.1 Basic Language Components 95
4.2 Coding Conventions 110
4.3 Reservation of Storage 112
Problems 125

Chapter 5
Binary Arithmetic and Related Machine Instructions **126**

5.1 The L, ST, and A Instructions 128
5.2 The LH, STH, and AH Instructions 130
5.3 The LR, STM, and S Instructions 131
5.4 The LM, AR, and SR Instructions 133
5.5 The LPR, LNR, and LCR Instructions 135
5.6 The MR, M, and MH Instructions 137
5.7 The SLDA, SRDA, D, and DR Instructions 140
5.8 The SLA, SRA, and SH Instructions 144
Problems 145

Chapter 6
Branching **147**

6.1 The LTR and BC Instructions 147
6.2 The TM Instruction 151
6.3 The C, CR, and CH Instructions 154
6.4 Assembler Language Subroutines 155
6.5 Address Modification and Loops 160
6.6 Indexing 164
Problems 167

Chapter 7
Decimal Operations **169**

7.1 Addition and Subtraction 170
7.2 Data Conversion 174
7.3 Multiplication and Division 179
7.4 Internal Movement of Data 183
7.5 The Compare Decimal Operation 193
7.6 The Translate Instructions 195
7.7 Editing Instructions 200
Problems 205

Chapter 8
Floating-Point Arithmetic **207**

8.1 Load, Store, Add, Divide, and Compare Instructions 208
8.2 The Load Positive and Subtract Instructions 212

8.3 Floating-Point Multiplication 214
8.4 Additional Load Instructions 216
Problems 224

Chapter 9
Logical Instructions **226**

9.1 The AND Instructions 228
9.2 The OR Instructions 231
9.3 The EXCLUSIVE OR Instructions 232
9.4 The STC and IC Instructions 234
9.5 Logical Addition and Subtraction 236
9.6 The LOAD ADDRESS Instruction 237
9.7 The Logical Shift Instructions 240
9.8 The Logical Compare Instructions 242
9.9 The EX Instruction 243
Problems 245

Chapter 10
Input/Output **247**

10.1 Basic I/O Processes 248
10.2 I/O and Related Instructions 255
10.3 Macro Instructions 260
Problems 264
Further Reading 266

Chapter 11
Additional Assembler Language Features **267**

11.1 Assembler Instructions 268
11.2 Advanced Assembler Instructions 271
11.3 Macro Instructions 275
Further Reading 287

Chapter 12
FORTRAN Programming **288**

12.1 Basic Coding Conventions 288
12.2 Numerical Variables and Arithmetic Expressions 290
12.3 Basic Sequence Changing 300
12.4 The DO Statement 304
Problems 311

Chapter 13
FORTRAN Input/Output **313**

13.1 Basic Output Operations 314
13.2 Basic Input Operations 317

13.3 Additional Format Possibilities 320
13.4 Tape Operations 327
Problems 329

Chapter 14
Additional FORTRAN Features **332**

14.1 Subprograms 332
14.2 The COMMON, EQUIVALENCE, and EXTERNAL Statements 343
14.3 Source-Program Statement Order 345
14.4 Complex Numbers 346
14.5 Logical Quantities 348
14.6 The G Format Specification 352
14.7 Other Features 353
Problems 355
Further Reading 357

Chapter 15
The COBOL Programming System **358**

15.1 Program Organization and Coding Conventions 359
15.2 Storage Allocation for Numeric Variables 364
15.3 Basic Computational Keywords 366
15.4 Sequence Changing in COBOL 370
15.5 Basic Input/Output Specifications 375
15.6 The OPEN and CLOSE Verbs 379
15.7 The Basic READ Statement 380
15.8 Figurative Constants 381
15.9 The EXAMINE Verb 382
15.10 Multiplication and Division 384
15.11 The WRITE Verb 385
Problems 392

Chapter 16
Data Description and File Organization **395**

16.1 The PICTURE Specification 395
16.2 File Processing 399
16.3 Testing for End of Page 404
16.4 The TRANSFORM Statement 405
16.5 Data Aggregates 409
16.6 Searching Direct Files 415
16.7 COBOL Subprograms 417
16.8 Special Routines for Error-Processing Declarative Sections 420
16.9 Other COBOL Features 422
Problems 424

Chapter 17
COBOL Advanced Features **426**

17.1 Sorting 426
17.2 Report Writing 431
17.3 Asynchronous Processing 440
17.4 Other Advanced Features 446
Problems 451

Chapter 18
Programming in the PL/I Language **453**

18.1 Basic Syntax and Coding Rules 454
18.2 Specifications for Data 457
18.3 Arithmetic Assignment Statements 462
18.4 Built-In Functions 463
18.5 Data Aggregates 465
18.6 Character and Bit String Handling 471
18.7 Loops and Decision Statements 475
Problems 484
References 485

Chapter 19
PL/I Input/Output Programming **486**

19.1 Stream-Oriented Operations 487
19.2 File Description 499
19.3 Record-Oriented Operations 503
19.4 Additional Input/Output Services 509
Problems 510

Chapter 20
Additional PL/I Features **514**

20.1 Organization of PL/I Programs 514
20.2 Additional Facilities for Interrupt Handling 522
20.3 Dynamic Allocation of Storage 524
20.4 Additional Storage-Handling Features 528
Problems 530
References 531

Appendix A
The ALGOL Language **532**

A.1 Basic Definitions 532
A.2 Basic Statements 538
A.3 Input/Output Operations 543
Supplementary Reading 549

Appendix B
Introduction to Job Control Language **550**

B.1 Coding Conventions 550
B.2 Job Identification 551
B.3 The EXEC Command 552
B.4 Device Assignment 557
Additional Reading 562
References 562

PROGRAMMING THE IBM SYSTEM/360

Chapter 1

Introduction to System/360 Organization

Although the IBM System/360 can be considered to be the definitive third-generation computer, an enumeration of the properties which motivate this designation is not at all straightforward. As indicated in the Preface, a number of its hardware features have been used on prior machines, and some of its basic software structures are direct second-generation carryovers. Yet it is undeniably a third-generation system in toto and it will be most consistent to treat it as such. Consequently, having used them as a frame of reference, we shall examine System/360's overall organization and major components without concerning ourselves further with generations.

1.1 SYSTEM ORGANIZATION

In previous contexts, the term "computer" or "computing system" usually referred to a particular type of processor and a number of peripheral devices which could be attached in various combinations. Although the size of the memory could vary within a given range by certain increments, the internal speed of a processor was characteristic of the particular type. In

sharp contrast to this concept, the term "System/360" is a much more collective one. It refers to an entire series of processors encompassing a very wide range of sizes, speeds, and other performance characteristics. (For purposes of this book, the term "System/360" will refer collectively to the following Models: 25, 30, 40, 44, 50, 65, 67, 75, 85, 91, and 195.) The idea of treating these processors as related members of a "family" of computing systems is much more than an organizational convenience. It is in fact the basic philosophy which guided the development of System/360 and pervades virtually every aspect of its design.

To gain some insight into the overall structure of System/360 it is very useful to make reference to the concept of architecture as used by Amdahl, Blaauw, and Brooks in one of their definitive articles on the systems design.[1]* The term is used to refer to the functional characteristics and properties of the system as viewed by a user of that system (a programmer). Implied in this usage is the idea that there need be no direct correspondence between a particular functional attribute and some physical system component which implements that operation. The necessity for using this concept provides us with some very important information about the basic system design approach. The supervisory software, which oversees and controls the flow of instructions and data, is not something that was superimposed on a predefined hardware complex. Both hardware and software were designed concurrently from the very beginning, with each being fully cognizant of the existence and properties of the other. Consequently, although certain functions may be implemented by component circuitry and others by system software, the user is generally unaware of which type of agent is operating on which facet of his job. Consequently the hardware and software for System/360 are inexorably interwoven and we shall not always be able to discuss them separately. Whenever it is necessary, we shall treat certain components from this "architectural" point of view, concerning ourselves primarily with their operating characteristics without defining what is done by hardware and what is done by software.

With this orientation in mind we shall undertake a brief examination of the system's underlying features. For our purpose we shall consider them in terms of three basic design objectives: performance improvement, increased compatibility, and universality. Although this type of categorization is somewhat oversimplified and may not accurately reflect a designer's approach, it does offer a convenient vehicle from the programmer's standpoint.

1.1.1 Improved Economy of Performance

The idea of evaluating the "performance" of a computing system has so many facets to it that one cannot speak of a "standard" measurement

*Superscripts indicate references listed at end of chapter.

or set of criteria. Some people think of performance in terms of speed of processing, their primary concern being "how long does it take for a machine to produce a set of results, given a particular information handling situation?" Others refer to "turnaround time" as an effective index of performance. It is argued that the benefits of fast and efficient execution are lost if there is an excessive time interval (also difficult to define) between submittal of a job and delivery of the results. Still others, using perhaps a more comprehensive outlook, maintain that the performance of a system is reflected in its time utilization (the number of useful or operating hours per day). Attempts to amalgamate these and other criteria into some kind of overall index usually result in a reference to "answers per dollar per month" or something similar.

Since an attempt to tie ourselves down to one criterion or another will lead to some confusion without enhancing the consistency of the discussion, we shall avoid this difficulty by treating the performance implications of a particular feature in its own context. In this way the specific aspect of performance most directly affected will be clearly defined and the reader can easily integrate it into the more general picture.

1.1.1.1 Improved Circuitry

One of the most dramatic developments in the short but impressive history of the electronics industry has been in the field of component miniaturization. System/360 represents the first commercially available series of IBM computers in which such components are used extensively. (Figure 1.1 compares such circuits to their functional equivalents in earlier configurations.) This type of circuitry (known as *solid logic technology*) provides several important advantages over its predecessors:

1. The time required to perform many internal operations has been substantially reduced. In many cases the signal transmission and switching rates are so high that the length of the paths between successive circuits can be an important factor in the timing. Thus by substantially reducing the size of a circuit or group of circuits it is possible to effect a corresponding reduction in the lengths of the paths which connect them.
2. Many of the miniaturized components and circuits are actually simpler than the ones they have supplanted. Consequently there is an accompanying increase in reliability which is reflected in a reduction in machine malfunctions.
3. The microminiaturization process is not confined merely to a reduction of existing circuits and component types. In certain instances, new types of components have been developed whereby a single unit provides the functional equivalent of several groups of components. This has simplified a number of manufacturing processes which in turn reduced equipment costs.

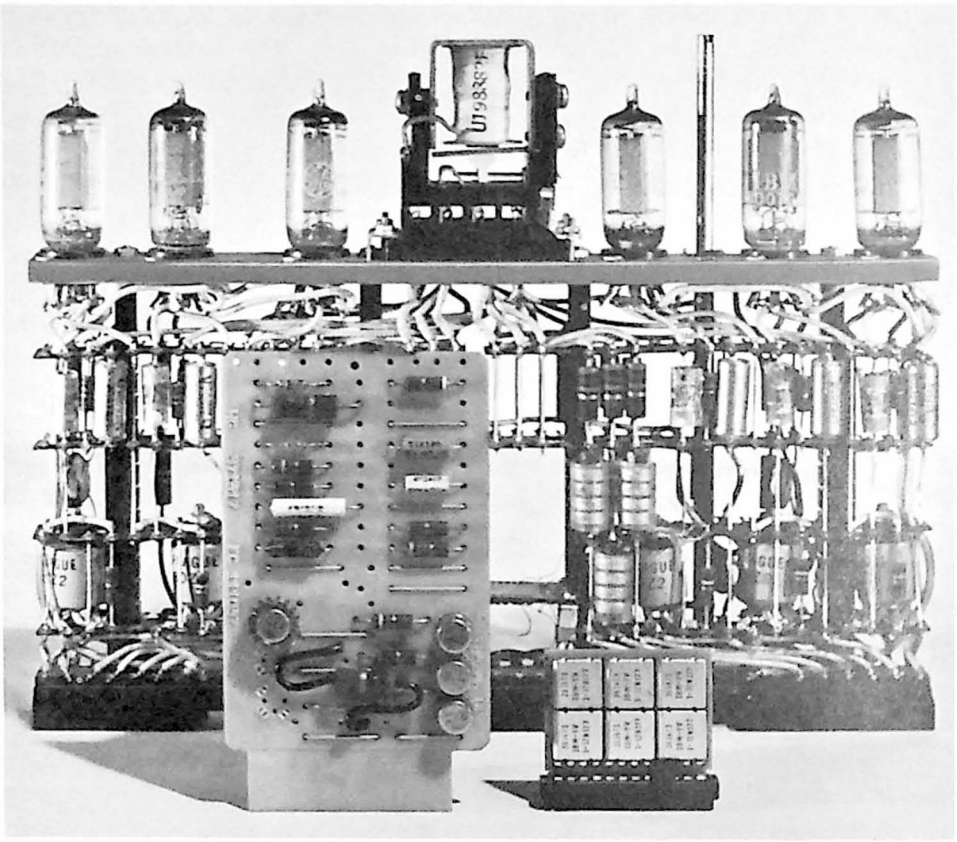


Figure 1.1 Comparison of Computer Circuits (Courtesy of the IBM Corp.)

Particular attention has been paid to problems of maintenance. As a result, the regular operating circuits of the System/360 are supplemented with an extensive set of circuits and components whose specific function it is to provide maintenance personnel with a comprehensive repertoire of checks and indicators for standard inspection and troubleshooting procedures. The inclusion of such hardware has greatly improved the maintainability of 360 processors and peripheral equipment without adversely affecting their performance when in normal use. In most cases, programmers and operators are unaware of the presence of these circuits.

1.1.1.2 Automatic Job Handling

One of the largest sources of inefficiency in computing systems has been the amount of time consumed between the termination of one job and the beginning of the next one. Although the introduction of supervisory software significantly reduced this unused time, there was still considerable slack because early versions of this type of software were super-

imposed on an existing hardware complex. In direct contrast, System/360 hardware and software were designed concurrently. Their close interaction has already been indicated with respect to the reduced correspondence between functional components and physical circuitry. Further evidence of this mutual dependence is seen by the fact that the basic 360 instruction set assumes the presence of an operating system. Indeed, it takes considerable perseverance, extensive knowledge of both the hardware and software, and a certain amount of skill to write a 360 program that can be loaded and executed on its own.

An important payoff of this hardware/software relationship is a further significant reduction of lost time between jobs by using a concept sometimes referred to as "nonstop operation." Under this procedural setup each job submitted to the system is treated as a member of a sequence of jobs. In a sense, the individual program is never in complete control of the system. When a job is completed or terminates because of some particular set of conditions, the machine does not come to a halt. Instead, the supervisory routines immediately begin preparation for the next job. Under certain circumstances these routines also select the next job from a waiting list. We shall say more about this later on.

This idea of nonstop operation has also changed the concept of the individual job. Under previous operating situations it was consistent to define a job as comprising the execution of a particular program (in conjunction with a given collection of data, as called for by that program). If the resulting output were to be used as input for another program, or if the original input were to be processed further by another program, this would be construed as another job and had to be set up on that basis. Under System/360's supervisory structure, what was previously called a job now becomes a *step* and can be a component of a multistep job. In many instances this may allow the continuous sequential execution of several procedures which otherwise would have to be performed as a number of separate jobs, strung out over several days.

1.1.1.3 Overlapped Operations

A continuing problem in the development of computer applications has been the wide gap between the electronic speed of a processor's internal operations and the rates associated with electromechanical input/output devices. As a result, many programs are described as being "I/O-bound" because their rate of execution is determined by the speed at which data can be transmitted to or from the processor rather than by the internal operating speed. The result of such a situation is, of course, that the processor spends a good deal of its time waiting for the completion of I/O operations, so that its speed advantages are partially negated.

The development of faster I/O devices has narrowed this gap somewhat,

but the effect has been offset, at least in part, by a concurrent increase in processor speeds. Another and much more effective approach has been to attach peripheral devices to a processor via one or more input/output *channels*. This allowed internal processing and data-transmission operations to proceed independently and therefore simultaneously. However, not all types of processors were equipped with the necessary hardware, and those that were required special software to make use of it.

All System/360 processors handle data transmission through I/O channels. Although the number of channels is optional, their use is inherent in the basic system design. Each channel operates like a separate and independent computer in that it has its own program whose execution is triggered by the supervisory software. This activity, which controls transmission via that channel, proceeds independently and does not affect internal processor operation.

Although the use of channel I/O goes a long way toward reconciling the speed discrepancy between data transmission and internal processing, the solution is by no means complete. In many instances the particular data-handling problem is such that it is logistically impossible to avoid situations in which the processor is waiting for the completion of I/O operations. This has motivated considerable work directed toward the development of systems which can accommodate several users with apparent simultaneity. Such multiple usage may take one of several forms, depending on the size of the configuration and the type of equipment it uses. One such setup (*time sharing*) is used in situations where the typical processing job involves sustained dialogue between user and system. That is, a certain amount of processing is requested, producing some output. The user, based on an inspection of these results, responds by defining the next portion of the procedure and requesting its execution. This type of operation is performed by a number of users, each working at an individual terminal which may be directly or remotely connected (in either case the connection is through an appropriate I/O channel). While one user is preparing his response to some output displayed on his terminal, the supervisory system goes on to perform a segment of another user's procedure. By taking advantage of the great difference between internal processing speed and user response time, this partial processing can be scheduled so that each user is unaware that other terminals are being serviced. A second type of operating mode (*multiprogramming*) can be implemented by having several jobs in memory at the same time. Processing on a particular job continues until the processor has to wait for completion of an I/O operation. At that point the supervisory routines turn their attention to another job and proceed to work on it. In more elaborate configurations an operating mode (*multiprocessing*) can be established in which several processors share a common memory, so that a number of jobs are actually worked on simultaneously.

A number of hardware and software components make such operations basically feasible on System/360. Crucial among these are the following:

1. 360 processors are equipped with a *memory protect* feature which allows the supervisory software to allocate and safeguard certain areas of memory for certain jobs. This makes it virtually impossible for the instructions in one job to write in or read from the portion of memory assigned to another. Furthermore, the software, protected from such possible damage under most contingencies, can change memory assignments at any time (even in the middle of processing) and make corresponding alterations in protection. Coupled with this feature is the basic ability to move a sequence of instructions from one area of memory to another (again, at any time) without affecting any aspect of performance. As we shall see in subsequent chapters, the basic addressing scheme is designed in direct conjunction with this relocating ability.
2. System/360 logical circuitry includes an elaborate set of components (known as an *interrupt system*) to switch from one job to another. The decision as to when to suspend processing on one job and when to begin (or resume) another is made by the supervisory routines based on a wide variety of criteria. These may be supplied by an I/O channel, a user's program, or by any of a number of sources. The significant point is that many of these signals are supplied and acted upon automatically, so that there is no need for the programmer to test for their occurrence. Furthermore, the status information required for proper resumption of a job at the point of interruption is generated and maintained as part of this automatic process.

1.1.1.4 Improved Efficiency in User/System Communication

System/360's integrated hardware/software design provides a good basis on which to build program language capabilities. At the symbolic level there is a very powerful assembly language which allows the programmer to take full and detailed advantage of the system's features. Added to this is an unparalleled repertoire of high-level languages which are or can be made available as part of the overall software package. Some of these, like RPG, FORTRAN IV, COBOL, PL/I, and ALGOL, have been developed by the manufacturer as part of the general 360 effort. Others, like TRAC, SNOBOL, LISP, and SIMSCRIPT, were developed by users for more specialized applications. Although these languages are handled by compilers which still represent early versions so far as the 360 is concerned (and significant improvements can therefore be reasonably expected), their performance, even at this stage, reflects the extensive experience on which these compilers have been built. In certain cases they contain operating features which represent significant departures from previous standard practices. For example, the PL/I compiler does not produce an intermediate assembly language program after analyzing the source statements. Instead, it codes directly in object form.

Another interesting feature that improves the efficiency of high-level

language program compilation and execution is the fact that standard computational routines are not generally shared by several compilers. For example, PL/I and FORTRAN IV use their own respective square-root routines, each being designed to operate most efficiently in the context of its own compiler.

Another basic feature that greatly enhances the ease of communication between user and system takes the form of a highly developed and versatile Job Control Language. This crucial software component is an organized artificial language in every sense of the word, having a well-defined vocabulary, syntax, and grammar. This organizational concept is in contrast to the extensive but relatively unintegrated set of job and control cards required by earlier software configurations. By embedding the various control functions within an overall structure, it is possible to facilitate the job of specifying various types of monitoring, scheduling, and allocating operations. Many of the commands which were previously included in programs have been shifted to the Job Control Language, with very powerful implications. For example, a program calling for input from a particular data file (regardless of the source language) can be used irrespective of the device on which that file is housed, as long as the data format remains consistent. One time the data may be on punched cards; another time, the file may be one of several on a reel of tape; in still a third instance the data may be stored as a multireel tape file; yet another occasion may call for data from a disk. For each case the particular file being used, as well as the device from which it is transmitted, are defined as part of the Job Control Language and not the program. By the same token, if a job consists of several steps and a particular data file is to be passed from one program to another in that job, the information defining that transmittal is supplied to the system by the Job Control Language.

1.1.2 Compatibility Among Computer Systems

Regardless of the systematic conveniences one may choose to impose on the history of computer development, it is difficult to disguise the fact that many of its aspects have been helter-skelter. Cohesive progress of the industry has been impeded by such inconsistencies as three different types of punched cards, several different magnetic tape widths, a number of different information codes for punched paper tape and various schemes for internal representation of data. Such differences did not always arise because different manufacturers were involved. Organizational, logistic, and operational difficulties were often encountered in two similar machines from the same manufacturer. To make things more complicated, the concept of separate machine types for "business" and "scientific" information handling, an idea which started as a useful sales device, grew into a way of life, making any attempts at standardization even more difficult. As a result,

many installations which outgrew their current computing systems were compelled to undergo the time-consuming and expensive processes of reprogramming their libraries and, in numerous cases, reorganizing their data files because of the different operational characteristics presented by the new machine.

One of the basic design intents in System/360 is to maintain a high level of organizational consistency across the range of processors in the series, thereby minimizing the effort required for upgrading a configuration when expanded requirements call for it. Consistent with the concept of "architecture" as discussed before, some aspects of this standardization are achieved by means of hardware while others are provided through software design, with the user being basically unaware of the distinction.

The complexity of the system, together with the extremely wide variety of possible configurations, make it unrealistic, with present technology, to speak in terms of total and complete compatibility. However, the salient aspects of the series' features that are compatible represent a substantial step toward that ideal.

1.1.2.1 Acceptability of Peripheral Equipment

In addition to allowing overlapped operations, the use of I/O channels in conjunction with versatile interface design makes it possible for most available I/O devices to be attached to all System/360 processors. This is accomplished by treating the peripheral device and its control as a single logical entity, regardless of whether or not the control is physically constructed as part of the device. Thus when the channel handles I/O instructions it does so by communicating with the device-controller combination. The I/O device is physically linked to the channel by a connector (known as the *I/O interface*) which is standard for all System/360 configurations. As a result it is possible to transmit data to or from any device with a standard repertoire of channel and processor instructions. Furthermore, this arrangement makes it unnecessary to restrict the usable devices to presently available types.

1.1.2.2 Common Organization of Memory

Although there is an extremely wide difference in capacity and speed between the two extremes in System/360 processors (we shall take a detailed look at this later in this chapter) the organization of memory (its division into bits, bytes, and words) is consistent over the entire range of processors. By the same token, the internal code used to represent numbers, letters, and other symbols is standard for the entire series. Thus a tape or other file of information generated by one member of the series will be recognized and properly interpreted by any other member.

1.1.2.3 Standard Instruction Sets

Regardless of size, speed, or internal organization, all System/360 processors are designed around the same basic set of instructions. This does not necessarily mean that all of the processors have the same "machine language." In fact, machine language in 360 context differs conceptually from that associated with earlier equipment. Instructions which we ordinarily think of as being in machine language (those comprising an object program) are not executed directly. Instead, these instructions are analyzed and fragmented into a larger number of microinstructions and it is those electronic commands which are actually executed. On some models this process is done by hardware, while on others it is performed by software components. In any case, the programmer is presented with a symbolic (assembly) language whose instruction formats and operation codes remain consistent, with very few exceptions, for all models of the series. Once a programmer has learned the assembly language for one model, he is already prepared to write programs for other models without substantial retraining.

1.1.2.4 Program Compatibility

The commonality of instructions over the range of System/360 processors carries over into the procedures by which these instructions are handled. Although various processor models may have different software structures equipped with their own assemblers, the object code language is standardized so that an object program prepared on one model is generally executable on another. This ability must be qualified by introducing the concept of "upward compatibility." This type of flexibility is possible only when both configurations are identically equipped (same memory size, same number of peripherals) or the original configuration (the one on which the source program was assembled) is smaller (has less memory, fewer peripherals, is a lower model number in the series) than the second configuration. Another general situation in which this type of compatibility cannot be assured exists when the program in question uses sequences of instructions which are dependent on the internal timing characteristics of the original configuration.

This idea of upward compatibility also extends to higher-level languages. Since certain very powerful language features depend on the existence of certain hardware/software capabilities which are present only in high model numbers of the 360, it becomes impractical to implement them for the entire series. To enable programmers to live with this situation and still provide upward compatibility, the versions of high-level languages and compilers for the various models are designed so that the instructions available in a particular language for one model represent a true subset of those available

for higher models. Thus, the object program resulting from a FORTRAN IV compilation on a 360/30 configuration can be executed on a 360/40, 360/50, and so forth, providing the other qualifications mentioned above are met.

1.1.3 Universality

The idea of separate machines and languages for "business" and "scientific" computer applications has become an increasingly expensive one to support. In addition, the accumulated experience in both areas has shown that there is considerable overlap both in terms of types of problems encountered and applicability of information-handling techniques. These two primary motivations, working in conjunction with improved hardware design and a number of other factors, have led to the organization of System/360 as a machine that is suitable for the widest variety of problems and applications. Thus, irrespective of whether the process is being handled by hardware or software, the programmer is presented with various types of instructions. When he finds it appropriate to set up variable-length fields and operate on them with decimal arithmetic, there are instructions to do so. Should he wish to deal with high-precision fixed-word lengths using an accumulator (as was characteristic of so-called "scientific" machines), there are instructions which allow him to operate as if the machine were organized that way. Similarly, the complete integration between hardware and software designs allows the inclusion of remote processing and multiprogramming capabilities as part of the basic system structure.

1.2 HARDWARE

As indicated earlier, it is often difficult (and not generally useful) to separate functional hardware and software features in System/360. Instead, we shall confine this discussion to an overview of the basic equipment components and how they interact.

1.2.1 General Organization

Although System/360 currently includes 11 models which encompass an imposing range of speeds, sizes, and performance capabilities, their basic organization is strikingly consistent. We have mentioned several compatible features such as standard data representation, I/O devices, and languages. This uniformity also holds with respect to the relationship between a central processor and the peripheral equipment attached to it.

All System/360 models use two basic types of I/O channels, known respectively as *selector* and *multiplexer* channels. Any device that receives data

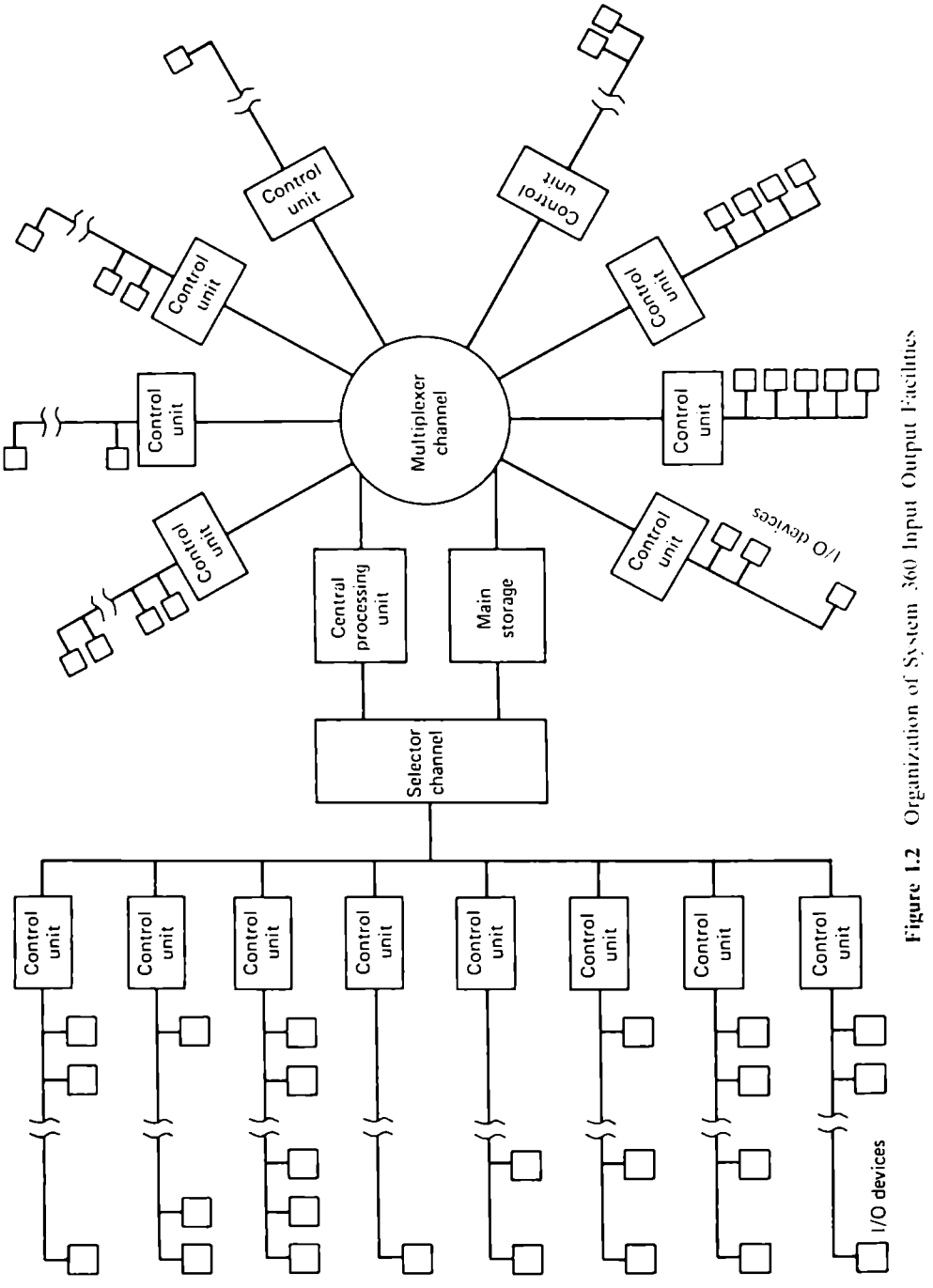


Figure 1.2 Organization of System 360 Input Output Facilities

from or transmits data to the central processor must do so through a channel. Although not conceptually apparent to the user, the connection between a peripheral component and a channel is not a direct one. The linkage is effected through a control unit. Each type of peripheral device requires a specially designed control unit, but once that unit is part of the overall configuration, several devices of a particular type may be attached to it. Furthermore, several control units may be attached to one channel. The system is expandable even beyond that point, since several channels may be attached to a central processor, the number depending on the model type. This organization is schematically shown in Figure 1.2.

1.2.1.1 Communication between Components

I/O channels may be constructed as separate physical units or may be housed within a central processor, depending on the particular model. In either case the channel is equipped with its own storage and is in effect a separate independent processor. It is this type of organization which allows extensive overlapping of I/O and internal operations. When a program being worked on by the processor requires an I/O operation to be performed, the processor signals the designated channel. Once this is done, the processor is free to go back to its own work. The channel takes over the data-transmission operations by bringing its own special program into its memory from that of the central processor and executing it. Data are then transmitted to or from the central processor without affecting the other activities going on at that time. In fact, the central processor can go ahead and trigger the execution of an I/O operation on another channel while the first one is still in progress.

With the addition of special equipment, it is possible to set up a line of direct communication between two channels without having to go through the central processor. This type of unit, called a *channel-to-channel adapter,* is regarded by each connected channel as a control unit. This type of arrangement becomes very useful if an installation often finds it necessary to move large quantities of data from one part of the processor's memory to another while other internal operations are going on.

One of the most dramatic features of System/360 organization is the ability to construct a multiple system in which several central processing units are interconnected. This may be done in three basic ways:

1. The slowest and most superficial type of connection can be set up by attaching a common I/O device to two central processing units.
2. Somewhat faster is the use of a channel-to-channel adapter as described before, but with the connections being made between a channel on each of the processors.

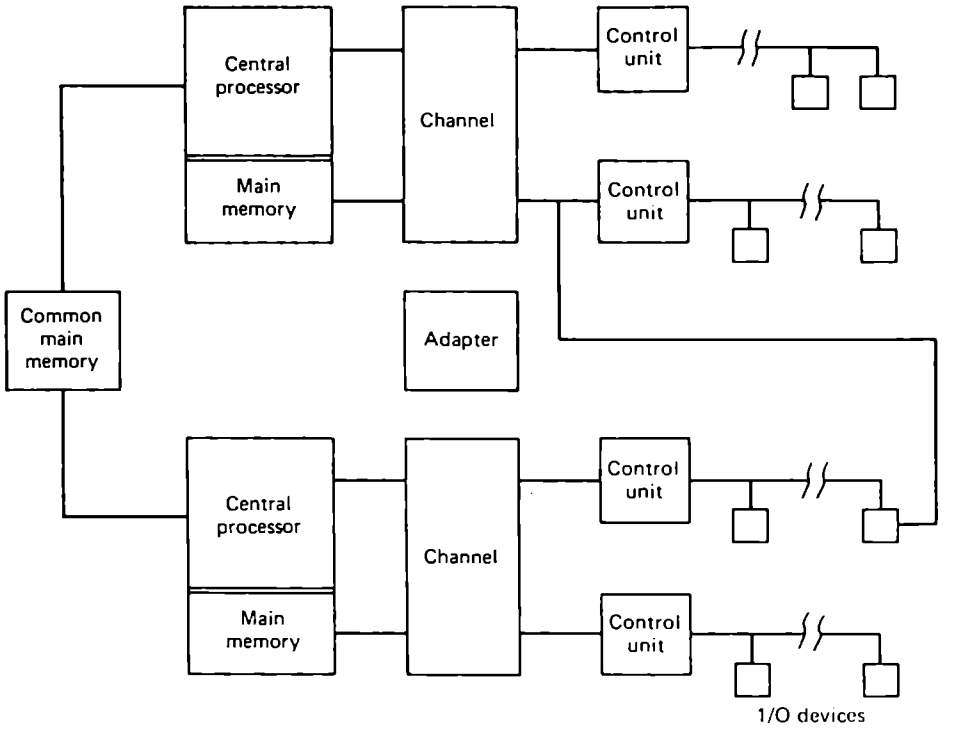


Figure 1.3 Schematic Representation of Communication Links Between Processors

3. The most intimate connection between processors is one in which several units share a common internal memory. However, this capability is currently restricted to the higher model numbers.

These various paths for intersystem and intrasystem communication are summarized in Figure 1.3.

1.2.1.2 Basic Properties of Selector Channels

This type of channel is intended for use with high-speed I/O devices such as magnetic tape drives, disks, and drums. Although many control units and, through them, many devices may be attached to a single selector channel (up to 256 devices through eight control units), only one device connected to that channel may operate at a given time. Transmission to or from another peripheral component on the same channel cannot be started until the signal is generated that operations have stopped on the previously selected one. Once a particular device is selected, the transmission rate may range from approximately 250,000 to 1,300,300 bytes per second, depending on the configuration type.

1.2.1.3 Multiplexer Channels

This type of channel is designed for attaching low-speed peripheral units (such as card readers, printers, card punches, typewriters, and other terminal devices) to a central processor. The channel circuitry itself is augmented by a sequential switch (hence the name "multiplexer"). Each step in this multiple switch (and there may be as many as 256 such steps) is termed a *subchannel,* to which may be attached a single I/O device. Depending on the types of devices connected in this manner, they may be assigned to one of as many as eight control units. Since the attached I/O devices are slow relative to the speed with which the switch can be set from one position to another, each subchannel appears to be operating as a separate channel and may be treated by the programmer as such. This type of operation is known as *multiplex mode.*

A second mode of operation on a multiplexer channel, known as the *burst mode,* occurs when a high-speed I/O device is attached to such a channel. Under those circumstances the switch does not step from one device to the next. Instead, only the one device can operate until all transmission is completed. Furthermore, this type of operation is not overlapped with internal processing on Models 25, 30, and 40.

Most models of System/360 will accept only one multiplexer channel. Models 44 and 67 are exceptions, with special types of channels being available for those configurations. Figure 1.4 summarizes the activity of multiplexer channels.

1.2.2 Central Processing Units

The term "central processing unit" is used conceptually rather than literally in this context because the functional components usually associated with a central processor do not always correspond to a particular physical unit across the entire 360 series. Our interest, for the present discussion, encompasses the internal memory, registers, arithmetic and control circuitry, and the control panel. In some models (as in Figure 1.5) all of these components, together with others, are housed in one unit; in others (such as the one shown in Figure 1.6) the control panel is installed in a console which is physically separated from the memory and other processor components. In either case the internal organization as presented to the user is consistent (with certain well-defined exceptions), as is the layout of the control panel.

The model number of a 360 system is dictated by the type of central processor at its heart. Two of the models in the series (360/44 and 360/67) deviate from the overall organizational consistency because of special objectives inherent in their design. Specifically, model 44 is oriented toward

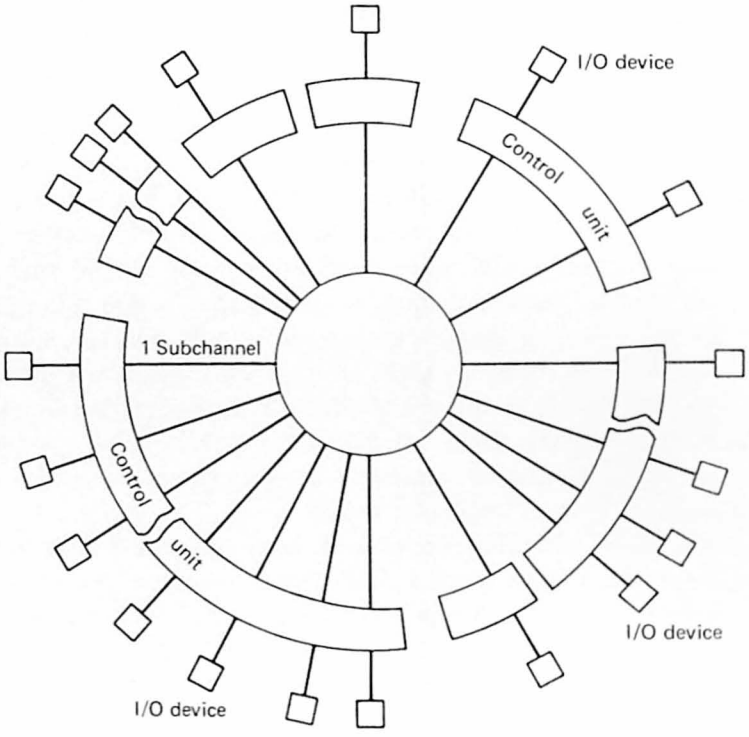


Figure 1.4 Schematic Representation of Multiplexer Subchannels

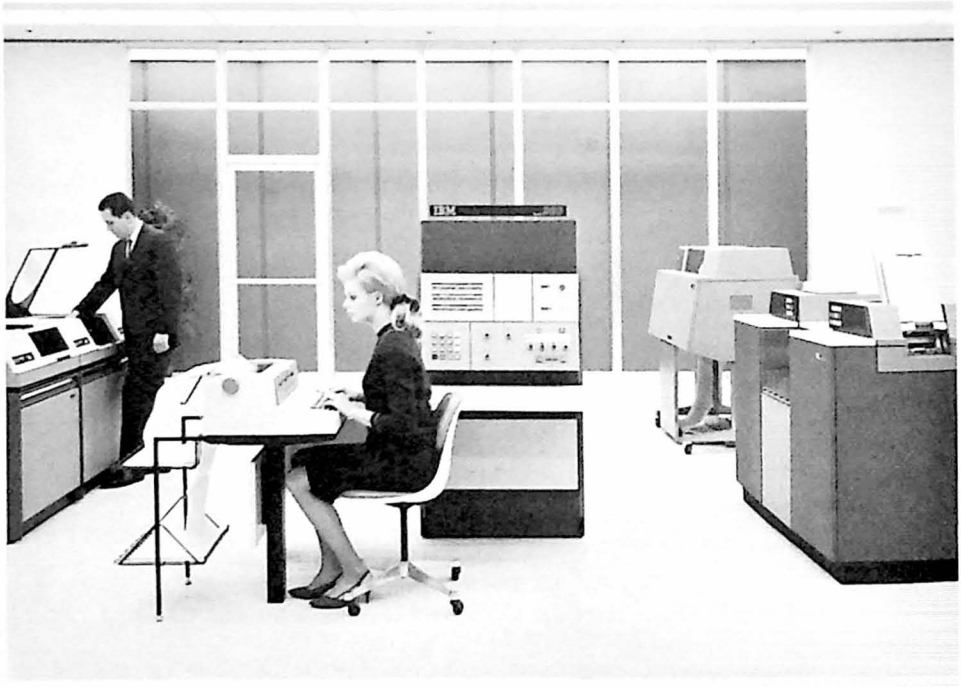


Figure 1.5 Model 360/25 (Courtesy of the IBM Corp.)

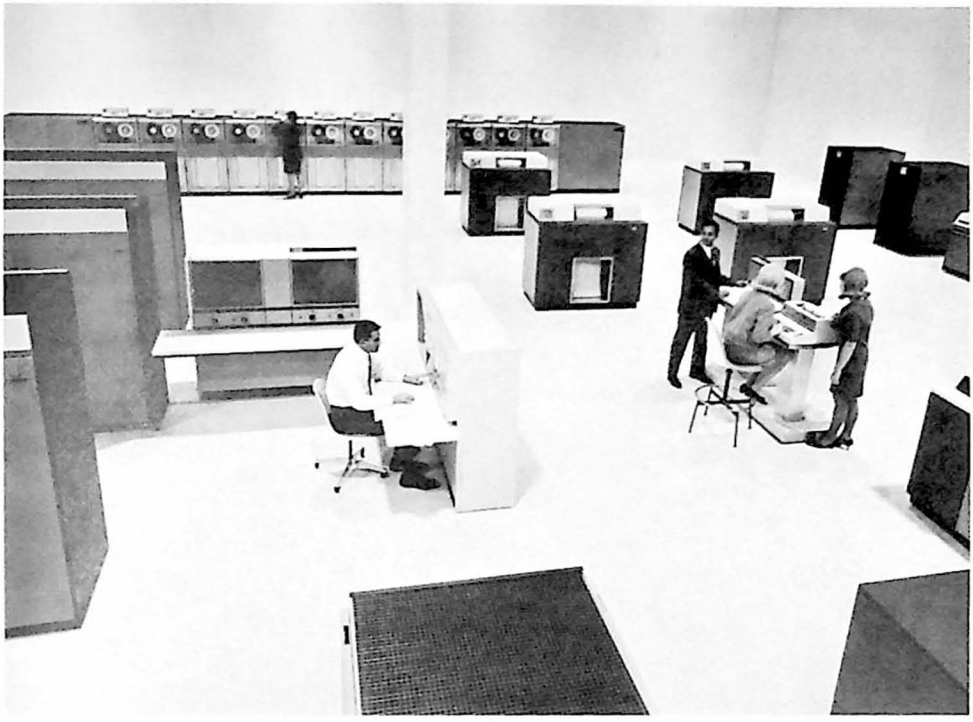


Figure 1.6 Model 360/85 (Courtesy of the IBM Corp.)

those installations at which the type of work predominantly involves floating-point operations on fixed word lengths, high-precision and data transmission to and from a variety of on-line devices, many of which may not be standard System/360 components. Model 67, intended for use in a time-sharing system, has many unique features which expand channel capabilities and facilitate the transmission of data to and from multiple remote devices and the connection of several processors to form an expanded multiprocessing configuration. We shall emphasize those features which are common to the series as a whole.

1.2.2.1 Processor Size

Consistent with the criteria for evaluating previous configurations, the size of a System/360 processor is defined exclusively by the amount of internal memory. Regardless of model, the smallest addressable memory unit in 360 processors is the individual byte (eight data bits plus one check bit), so that all sizes are given as numbers of bytes. To increase the flexibility of operations and provide the user with an opportunity to exercise some control over the economics of his facility, each model is available in a range of sizes and there is considerable size overlap among the various models. Consequently, a standard set of letters has been defined to represent various

TABLE 1.1 SIZE DESIGNATIONS FOR SYSTEM/360 PROCESSORS

Size in Bytes	Shorthand Notation	Size Code
8,192	8K	C
16,384	16K	D
24,576	24K	CD
32,768	32K	E
49,152	48K	DE
65,536	64K	F
131,072	128K	G
262,144	256K	H
524,288	512K	I
786,432	768K	HI
1,048,576	1024K	J
1,310,720	1280K	HJ
1,572,864	1536K	IJ
1,835,008	1792K	HIJ
2,097,152	2048K	K
3,145,728	3072K	JK
4,194,304	4096K	L
5,242,880	5120K	JL
6,291,456	6144K	KL

sizes and these are used as prefixes to qualify the model numbers. These designations are summarized in Table 1.1 and the available model sizes are given in Table 1.2. Since the various sizes are all multiples of two, it is somewhat awkward to refer to the exact magnitudes. Instead, a shorthand notation is used in which 1K represents 1,024 bytes. Thus a size H memory refers to 256K or 262,144 bytes. Note that although 6,291,456 bytes is listed as the largest available on a 360 processor, this is a current figure and does not represent a limiting size. The maximum possible memory size is dictated by the type of addressing used in the system. Since the 360 series uses 24 bits to specify a machine address, it is possible to specify a quantity as large as 2^{24} (16,777,216).

TABLE 1.2 STORAGE SIZES FOR SYSTEM/360 PROCESSORS

Model	Available Sizes
360/25	C, D, CD, E, DE
360/30	C, D, E, F
360/40	D, E, F, G, H
360/44	E, F, G, H
360/50	F, G, H, I
360/65	G, H, I, J
360/67	H, I, HI, J, HJ, IJ, HIJ
360/75	H, I, J,
360/85	J, K, JK, L
360/91	J, K, JK, L, JL, KL
360/195	J, K, L

The capacity of internal memory (for models 50, 65, and 75) can be expanded even further through use of large-capacity storage (LCS). This additional memory (also known as *bulk core*) is organized just like the regular processor's storage in that each byte has an individual address, with the address of the first byte of LCS following that of the last byte of processor storage. Its functional characteristics, however, differ from those of the basic processor's memory in that it is slower. Both types of storage are collectively referred to as *main memory*.

LCS units are constructed in blocks of 1,048,576 bytes (termed one megabyte) or twice that. Such units may be combined up to eight megabytes (8,388,608 bytes). Thus LCS can serve as a repository for large amounts of data or instructions which were stored on auxiliary devices (such as tape, disk, or drum) in previous configurations. Although more expensive than such peripheral components, LCS can often justify its cost based on elimination of time-consuming I/O operations.

1.2.2.2 Processing Speeds

The basic speed of System/360 processors is expressed as the amount of time (usually reported in microseconds and referred to as *access time*) required to obtain information from or place it into a specified memory location. It is difficult to compare speeds for all processors on this basis because the number of bytes handled during an access cycle is not consistent for the various models. As seen in Table 1.3, the smaller models access one or two bytes at a time, with the larger ones accessing eight bytes during a single cycle. This does not mean that a Model 50, for example, is able to access as many as four bytes at a time; it always accesses that many.

These figures may be misleading for still another reason. As reported, they represent the physical access time, the majority of which is taken up

TABLE 1.3 ACCESS TIMES FOR SYSTEM/360 PROCESSORS

Model	Number of Bytes Accessed per Cycle	Access Time, microsec
360/25	2	1.8
360/30	1	1.5
360/40	2	2.5
360/44	4	1.0
360/50	4	2.0
360/65	8	0.75[a]
360/67	8	0.75[a]
360/75	8	0.75[a]
360/85	16	1.04[a]
360/91	8	0.75[a]
360/195	8	0.75[a]

[a] Interleaving of access cycles in these models substantially reduces effective access times.

by the performance of certain housekeeping duties at the particular memory location after data have been received or read out. Since the processing circuitry is not involved in this housekeeping, it is in effect waiting for the memory circuits to finish their business before it can perform the next access cycle. Special hardware is included in some System/360 processors to allow accesses to be made to a sequence of locations without waiting for the entire cycle to be concluded. This process, known as *interleaving*, reduces the effective access time to a fraction of its physical value. The extent of interleaving is determined by the number of locations among which accesses may alternate. Depending on the particular processor model, this may be anywhere from two to sixteen.[2]

The speed of Large Capacity Storage is not a function of the model to which it is attached: access time is constant at eight microseconds. However, the number of bytes accessed during that time is consistent with that of the primary memory.[3]

System/360 processors are equipped with hardware components which allow stored information to be protected from mishandling. With this circuitry it is possible to divide main memory into blocks of 2,048 bytes (not necessarily contiguous). Each block can then be protected in two separate ways. *Store protection* prevents the contents of storage from being changed by program errors or input from external devices. This makes it possible for a number of programs to be stored at the same time without danger of intrusion. The second type of protection, known as *fetch protection*, prevents the reading of information from one or more particular storage blocks.

Store protection is standard on all models above 40 and optional for others. Fetch protection is standard on Model 65 and higher and can be added to a Model 44. It is possible to use store protection without fetch protection, but not vice versa.

1.2.3 Peripheral Equipment

System/360 processors can be equipped with an extensive array of I/O equipment offering a wide range of transmission rates and auxiliary storage capacities. By using a standard interface design, this versatility is extended to include components yet to be introduced.

1.2.3.1 Card Readers and Punches

Several types of these devices have been designed in an effort to accommodate the great diversity of work loads at various types of installations. Toward this end there are units with different operating speeds; some models combine reading and punching facilities in a single physical unit, while others offer one or the other operation. These and other characteristics are summarized in Table 1.4.

TABLE 1.4 BASIC OPERATING CHARACTERISTICS OF SYSTEM/360 CARD READERS AND PUNCHES

Model Number	Cards per Minute Read	Cards per Minute Punched	Buffered	Control Unit	Maximum Number of Devices on Control Unit
1442N1[*1]	400	91 (average)	No	Built-in	1
1442N2	Punch only	91 (average)	No	Built-in	1
2501B1[5]	600	Read only	No	Built-in	1
2501B2	1,000	Read only	No	Built-in	1
2520B1[*6]	500	500	For punch	Built-in	1
2520B2	Punch only	500	Yes	Built-in	1
2520B3	Punch only	300	Yes	Built-in	1
2540[*7]	1,000	300	Yes	Separate (2821)	4

[*] These units are arranged so that the same card passes the reading and punching mechanism, thus enabling the system to add punches to a prepunched card.

1.2.3.2 Line Printers

An installation may choose among several printer models depending on its volume and type of output. In addition, these printers may be equipped with various types of character sets, including custom designs. Unlike some of the card readers and punches, all of the line printers are buffered so that internal processing and printing can proceed simultaneously. Table 1.5 presents a summary of their operating properties. Since the maximum printing speed is dictated by the number of different characters on the chain as well as the physical construction of the device, such figures cannot be represented very accurately. Hence the information given in the table reflects maximum speeds for "typical" print chains.

1.2.2.3 Cathode-Ray Tube Display Devices

System/360 hardware includes two basic types of cathode-ray output devices which can be attached to I/O channels for all processors in the series. While functioning basically as visual display units, these devices may be equipped with a wide range of accessories which can transform them into complete inquiry stations and control consoles.

The simpler of the two devices is the 2260 display station. In its basic form it provides a 4 × 9-inch viewing screen on which as many as 960 characters may be displayed at one time using an alphabet of 64 different characters. As with other output devices, the contents and format of the display are controlled by the programmer.

This type of unit is attached to an I/O channel by means of a control unit (2848) which includes its own memory. This allows overlapped operation with the central processor in the following way: since the output is

TABLE 1.5 SUMMARY OF OPERATING CHARACTERISTICS FOR SYSTEM/360 LINE PRINTERS

Model Number	Maximum Number of Columns	Maximum Speed (Lines per Minute)	Control Unit	Maximum Number of Printers on Control Unit
1403-2[a]	132	600	Separate unit (2821)	3
1403-3	132	1,100	Separate unit (2821)	3
1403-7	120	600	Separate unit (2821)	3
1403-N1[a]	132	1,100	Separate unit (2821)	3
1404[b]	132	600[b]	Separate unit (2821)	2
1443-N1[c]	132	600	Built-in	1
1445-N1[c]	113	525	Built-in	1

[a] This unit operates like the 1403-3; its external design has several improved features, including additional sound absorption.

[b] This unit has a dual function in that it can be used to print on punched cards (maximum rate is 800 cards per minute). It is also equipped to read a card and print some of its contents on that card.

[c] This unit is equipped to print with magnetic ink using a set of characters employed in banking transactions.

digital, it must be repeatedly displayed on the screen in order to give the illusion of a stationary picture. Once the output to be displayed is transmitted from the central processor to the control unit memory (more commonly called the *buffer*), reinforcement of the image is handled strictly by the control unit and the processor can go on to other operations.

The functions of this display station can be expanded to include those of an inquiry terminal by adding a numeric or alphameric keyboard which can be used to transmit messages to the processor. Before such data are sent, they are displayed and can be modified from the keyboard. By the same token, the 2260 user may request some specified data from the central processor. He can then edit the resulting display from his keyboard and send it back to the processor's storage.

The 2250 display unit (Figure 1.7) is larger, offering a 12 × 12 inch display screen. More importantly, it offers a wide variety of display and input capabilities. In addition to providing character output, its display surface is divided into roughly one million distinct points, each with its own set of coordinates which can be cited by the programmer. These points are sufficiently close to each other to provide good representations of continuous

lines and curves, thus permitting the display of drawings, diagrams and other graphical output.

A very powerful component of this device is the light pen. This instrument can be used to point to a particular character, line or point on the display, thereby identifying it to a program. This facility enables the user to "draw," "write," or "erase" on the screen and have the result of his action manifested as appropriate alterations in memory.

The 2250 unit is equipped with a keyboard which can be used in conjunction with the light pen or as a separate input/inquiry unit. The keyboard may be either a *programmed function* keyboard, in which case each of its 32 keys represents a particular type of activity specified by the user as a routine. Thus, for example, by pointing the light pen to a line on the screen and depressing a given key, a program can be activated to move the line closer to the top of the screen and make the corresponding changes in memory. The other type of keyboard is similar to that of a typewriter (*alphameric keyboard*) which operates similarly to that for 2260 display unit. The 2250 may also be equipped with a device that produces a copy on paper of images displayed on the screen.

Some of the basic operating characteristics of these CRT devices, together with those of their associated control units, are summarized in Table 1.6.

Figure 1.7 Model 2250 Display Unit with Light Pen (Courtesy of the IBM Corp.)

TABLE 1.6 SUMMARY OF CRT DEVICE OPERATING CHARACTERISTICS

Model	Control Unit	Maximum Number of Devices on Control Unit	Maximum Buffer Unit	Transmission Rate Between Buffer and Processor, Bytes/sec	Maximum Transmission Rate Between Buffer and Screen, Bytes/sec	Maximum Number of Characters on Screen
2250-1[11]	Built-in	1	8K(8192 bytes)	238,000	59,500	3,848 or 1,715[b]
2250-3[12]	2840-2	4	32K(32,768 bytes)	500,000[a]	59,500	3,848 or 1,715[b]
2260[13]	2848-1	24	240	2,560[a]	28,800	240
2260	2848-2	16	480	2,560[a]	28,800	480
2260	2848-3	8	960	2,560[a]	28,800	960

[a]These are the transmission rates when the unit is locally (i.e., directly) connected to its control unit (2,000 feet is the limit for this type of connection). When the control unit is connected remotely to the I/O channel and data must be transmitted over telephone lines, these rates decrease.
[b]The number of characters depends on which of the two available sizes is being used for a particular display.

1.2.3.4 Magnetic Tape Units

Since magnetic tape is a very commonly used data-storage medium, the System/360 line of equipment includes transport units which cover a wide range of performance characteristics. In addition to providing improved tape-handling ability by incorporating new hardware technology, these units are designed to accept tape files which were created on many of the earlier configurations, thus avoiding the arduous task of regenerating them. This is done by equipping the tape units with the ability to recognize characters using seven tape tracks for representation (nine is the standard for System/360) and stored at densities of 200 and 556 characters per inch of tape length (versus 800 or 1,600).

Although a complete description of these transport units' operating characteristics is inappropriate here, certain new features should be noted. In previous designs the tape reel had to be traveling forward (i.e., away from the beginning of the tape) for information to be transmitted from it to the processor. With these units it is possible to perform such reading while the tape is moving in either direction. (This is known as the *read backwards* feature.) Thus if the particular program presents the opportunity for it, the programmer may have some additional reading done while the tape is rewinding, thereby effectively using the time which was previously lost. A second important feature is concerned with improved file-protection capability. In earlier systems it was possible to protect a tape file (making it impossible to write on that tape) by removing a special plastic ring from the reel. Such protection can now be specified as part of a program, making it possible to install and remove it without physical interruption. Similarly, the density at which a particular tape is to be written or read was previously defined by a physical setting on the tape drive or processor control panel. Now this specification can be made in a program. Additional operating characteristics are summarized in Table 1.7.

TABLE 1.7 SUMMARY OF MAGNETIC-TAPE DRIVE CHARACTERISTICS

Model	Control Unit	Maximum Number of Drives on A Control Unit	Storage Density, Bytes per Inch	Number of Tracks	Tape Speed, In./Sec	Maximum Transmission Rate, Bytes/Sec
2401-1[14,15]	2803 or 2804	8[a]	200, 556, 800	7 or 9	37.5	7,500, 20,850, 30,000
2401-2	2803 or 2804	8[a]	200, 556, 800	7 or 9	75	15,000, 41,700, 60,000
2401-3	2803 or 2804	8[a]	200, 556, 800	7 or 9	112.5	22,500, 61,950, 90,000
2401-4	2804	8[a]	800, 1,600	9	37.5	30,000, 60,000
2401-5	2804	8[a]	800, 1,600	9	75	60,000, 120,000
2401-6	2804	8[a]	800, 1,600	9	112.5	90,000, 180,000
2415-1	Built-in	2	200, 556, 800	7 or 9	18.75	3,750, 10,425, 15,000
2415-2	Built-in	4	200, 556, 800	7 or 9	18.75	3,750, 10,425, 15,000
2415-3	Built-in	6	200, 556, 800	7 or 9	18.75	3,750, 10,425, 15,000
2415-4	Built-in	2	200, 556, 800, 1,600	7 or 9	18.75	3,750, 10,425, 15,000, 30,000
2415-5	Built-in	4	200, 556, 800, 1,600	7 or 9	18.75	3,750, 10,425, 15,000, 30,000
2415-6	Built-in	6	200, 556, 800, 1,600	7 or 9	18.75	3,750, 10,425, 15,000, 30,000
7340-3[b16,17]	2802	8[a]	1,511, 3,022	10	112.5	170,000, 340,000

[a]In special cases up to 16.

[b]This unit uses 1-in.-wide tape, compared to 1/2 in. for all of the other units.

Of special note is the 7340 tape unit known as the *hypertape drive.* This device provides storage densities and transmission rates that are considerably higher than those available with other models. Because of the special design requirements commensurate with this increased performance, the tapes produced by these units are incompatible with any others. In addition to the unique characteristics shown in the table, the tape width is different (1 inch versus .5 inch for all other System/360 tape drives) and it is mounted in a special sealed loading cartridge. Furthermore, it is possible to reduce

Figure 1.8 Model 2311 Disk Drive (Courtesy of the IBM Corp.)

operator intervention by placing a second cartridge in a "ready" position while the first is being processed. An instruction in the user's program can then cause the first cartridge to be unloaded, followed by the automatic loading of the second cartridge. That cartridge may now be processed normally and the operator can remove the first cartridge and install the next one in the ready position while processing continues.

1.2.3.5 Direct-Access Devices

In addition to magnetic tape, the array of System/360 peripheral units includes a number of magnetic storage devices previously called random-access devices and presently known as *direct-access* devices.[18] Although their method of storing and representing information is not basically different from that used for magnetic tape, the design of these devices is such that information at any location on the storage surface may be obtained by accessing it directly (without searching adjacent storage areas). This is in contrast, of course, to the sequential nature of tape, which cannot be conveniently partitioned into adjustable portions. Consequently the time required to obtain the contents of a particular location on a direct-access device is less dependent on which location it happens to be then in the case of magnetic tape.

These devices use three basic physical designs. Disk drives (such as the one in Figure 1.8) provide storage on top and bottom surfaces of a series of concentric rotating disks coated with magnetic material. On other direct-access devices the information is stored on the outer surface of a rotating drum (Figure 1.9). In contrast to these types, versions of which were available on earlier IBM configurations, System/360 also provides a direct-access device known as a *data cell* composed of a rotating assembly of ten replaceable modules (Figure 1.10). Each of these contains twenty strips of magnetic tape and each individual strip is divided into addressable portions.

A summary of direct access device operating characteristics is given in Table 1.8.

1.2.3.6 Other Peripheral Equipment

A variety of additional devices is currently available for use as System/360 peripherals.[2] Input units include devices that accept information from punched tape and those which handle specially designed cards and pages (mark-sense readers). Others are built for data printed with magnetic ink. Several types of optical readers are available for transmitting a variety of printed and typewritten input, including register tapes and output from the 1403 line printer. A special device for reading certain hand-printed characters can also be obtained.

Additional output peripherals include units for printing with magnetic

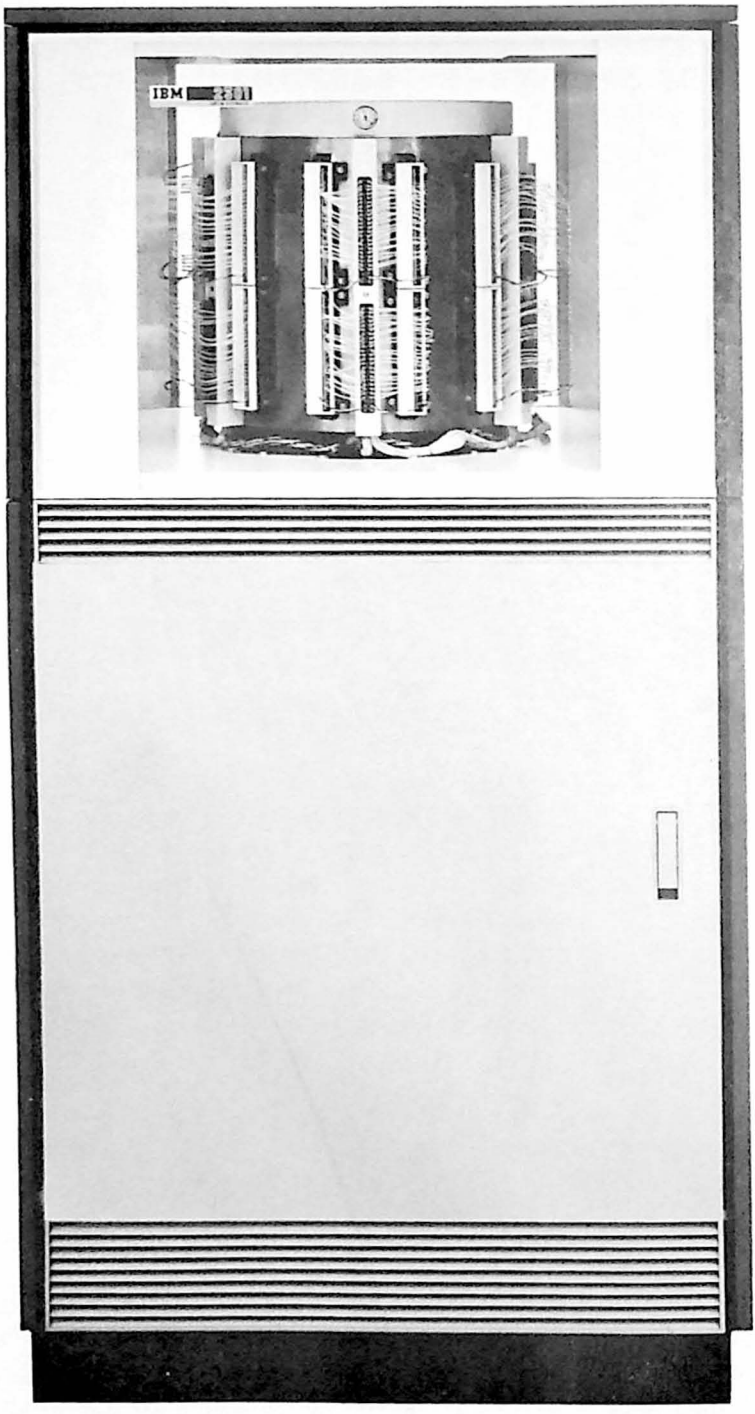


Figure 1.9 Model 2301 Drum Storage Device (Courtesy of the IBM Corp.)

Figure 1.10 Model 2321 Data Cell (Courtesy of the IBM Corp.)

TABLE 1.8 SUMMARY OF DIRECT ACCESS DEVICE OPERATING CHARACTERISTICS

Model	Storage Capacity, Bytes	Control Unit	Maximum Number of Devices Per Control Unit	Access Time Range, Sec	Data Transmission Rate, Bytes Sec	Device Type
2302-3[a,26]	112,790,000	2841[b]	4	.050-.180	156,000	Disk
2311[c,19]	7,250,000	2841	8	.025-.135	156,000	Disk
2314[20]	233,400,000[d]	Built-in	1	.025-.135	312,000	Disk
2303[21]	3,910,000	2841	2	0-.180	312,000	Drum
2301[22,23]	4,000,000	2840	4	0-.180	1,200,000	Drum
2321[24,25]	4,000,000	2841	8	.0001-.650	55,000	Data cell[e]

[a]Model 2302-4 is the same as 2302-3 except that it is a double unit, containing two storage modules. The 2841 control unit can accommodate two 2302-4's.

[b]One 2841 control unit can accept up to eight devices in any combination, with certain restrictions regarding the use of 2302 disks and 2303 drums.

[c]The 2311 storage medium (i.e., the disk pack) is removable and replaceable.

[d]The data cell consists of ten individually replaceable modules, each accommodating 40,000,000 bytes.

[e]This capacity is equally divided among eight independent disk drives, each with removable pack. There is also a ninth drive available as a spare when one of the other drives is not available for use.

ink and a series of devices which produce verbal output by combining words from a prerecorded addressable vocabulary under program control. Various types of graph plotters, although not manufactured by IBM, can also be attached to System/360 channels.

In addition to these peripherals, there are many types of keyboards, printers, cathode-ray tubes and other devices for use as terminals in data-communications systems. These units are designed for remote input/output via telephone lines or other carriers.[27,2]

1.3 SOFTWARE

Although the distinction between what is and what is not computer hardware is not always clear, our discussion has been sufficiently general to avoid this difficulty. However, a cohesive scrutiny of software, regardless of its depth, requires some type of definition as a frame of reference.

Because of the organizational philosophy which governs System/360 design, we shall find it best to use the term "software" in its most general context. That is, we shall consider it as a broad canopy encompassing *all* programs and routines. Although this type of definition is used by some people, it is not universal and is in direct contrast to the alternative concept which treats software as consisting of those programs and routines that make the hardware's capabilities more accessible to the user. This more restrictive

definition separates utility, supervisory, and other so-called "service" or "systems" programs from applications programs, which produce output of direct interest to the user.

The reason for adopting the less selective view of software lies in the structure of System/360's operating system. (There are actually several operating systems for the various models but we shall speak generically of a "single" one at this point.) The fundamental design concept divides System/360 software into two basic categories: Those components that control and oversee the system's functions and those that process data submitted to the system. It is the second category which forces a broader definition of software, since any program or routine that processes data is handled in the same way, irrespective of its origin. A compiler, which processes a set of source language instructions, and a user's program, which operates on his data, are functional equivalents to the control components. Thus, although the discussion of processing programs will emphasize compilers and other software components usually supplied by the manufacturer, it should be understood that applications programs are to be considered in the same context.

1.3.1 Control Components[2]*

Although it constitutes a substantial oversimplification, we can say that the basic function of an operating system is to provide a variety of functions and services which allow users to realize effective utilization of a computing system with maximum convenience. The fulfillment of this purpose is closely tied in with the ability of the operating system's control components to provide efficient manipulation of all of the information (programs and data) that constitutes the system's work load.

From the user's and application programmer's viewpoint, the work load under a System/360 operating system consists of a series of *jobs*. Each job represents some type of information-handling procedure which requires the execution of at least one program but may entail the use of a sequence of several programs. When the latter is the case, that job is said to be composed of several *job steps*. In some instances it is up to the user to define each step. A typical example is the sequence of operations in which a particular program is used to process some data, with the resulting output serving as input to a second program which produces the final results. Under System/360 control programs this is considered as a single job consisting of several steps. Some steps are explicitly described by the user as part of his job specifications. For certain situations there are mechanisms within the

* Further information on control programs can be found in System/360 manuals dealing with each type of operating system. For Basic Programming Support, see Refs. 28 and 29; for Basic Operating System, Refs. 30 and 31; for Tape Operating System, Refs. 29, 32, and 33; for Disk Operating System, Refs. 29, 32, and 34; and for Operating System, Refs. 35, 36, and 37.

control programs that will divide a job into its component steps automatically. This occurs, for instance, when a source program is submitted for compilation, followed by execution of the produced object program. Although the programmer treats this as a one-step job, its actual implementation involves the use of three separate processing programs (the last of which is the user's executable program). With this job concept in mind we can now examine the major duties of the control programs by dividing them into three basic categories:

1. *The Job Scheduler*

This integrated set of routines is designed to handle a sequence of jobs so that there is a minimum amount of operator intervention required between jobs. To do this, the scheduler reads and analyzes the job description submitted by the user, allocates the required input/output devices for each job step, initiates and terminates program execution, and keeps a detailed record of activities.

The repertoire of System/360 software includes several types of job schedulers so that each installation can incorporate the one which matches its requirements most closely. The simplest type is the *batch* scheduler, which is designed to handle one job at a time. When sufficient hardware capabilities are available to support it, a *parallel* job scheduler can be used. This version controls the operation of several job steps which may or may not belong to a single job. A third type of scheduler (the *priority* scheduler) is designed to rearrange the order in which jobs are processed based on priority and/or availability of facilities. Thus a particular job can be performed with facilities which are not required by another job already in process (e.g., the performance of I/O operations while the other job uses the central processor). The priority scheduler will start that job even though there may be one waiting with higher priority.

2. *The Master Scheduler*

This component presents a major communication link between the operator and the system. When acting as a receiver, it can accept messages from the operator to start new operations, change the status of input/output units and alter the disposition of jobs by passing messages to the job scheduler. It also provides the vehicle for transmitting messages from the system to the operator relating the status of various components within the system. As is the case with the job scheduler, there are several different types of master schedulers, each with its own range of capabilities.

3. *The Supervisor*

Primary coordination of System/360's activities is handled by the supervisor, which can be thought of as the control center for the entire operating system. Any time there is an interruption, whether it is triggered automatically (as occurs at the end of I/O), by a user's program, or by some other operating system component, control is immediately turned over to the supervisor, where the subsequent activity is determined.

Since the operating system contains a large number of control programs many of which are not required concurrently, their permanent storage in main memory would be extremely wasteful. Consequently, the system is structured so that the supervisory routines (which constitute a relatively small portion of the overall system) are kept in memory. Through the use of special input/output instructions available only to the operating system, the supervisor is able to regulate the traffic of other control programs between main and auxiliary storage. These and other *privileged instructions* allow the supervisor to allocate storage for these programs and to protect them as necessary from inadvertent erasure. Multiprogramming and remote processing operations are also controlled by the supervisor.

These three major components are supplemented by a number of routines which, while not easily categorized, provide important additional control functions.

While it is convenient to treat the control programs organizationally in terms of a job scheduler, master scheduler, and supervisor, we must examine them from another viewpoint in order to become more fully aware of the breadth of operations under their charge. To do this we shall take a closer look at the handling of jobs (usually referred to as job management), as well as two very important support activities handled by the control programs: task management and data management.

1.3.1.1 Task management

One of the most complex control functions in an operating system is the allocation of resources to get a particular job done. In order to carry this out most effectively, the control programs partition each job step into one or more *tasks*—a task being the smallest piece of work which requires some type of system resource, such as control of the central processing unit, a specified amount of main memory, a particular I/O device, a given program, etc. Once the job scheduler has started the execution of a job step, its component tasks are turned over to the task management programs which determine what resources are needed and go about getting them. In a simple (single task) system these resources are provided as requested. For more extensive facilities, it is possible to include management programs which allow the system to work on more than one task at a time (*multitasking*). Under those circumstances the tasks being worked on at a given time must compete for resources based on their respective priorities.

As is the case with the identification of certain types of job steps within a job, the division of a job step into its component tasks often produces additional requests for resources of which the programmer is not directly aware. For example, when a programmer writing in some high-level source language makes use of one of the routines built into that language, a separate task must be assigned to take care of the allocation of storage for that

routine. This implicit use of main memory is in contrast to an explicit type of allocation in which the programmer specifically asks for a certain amount of storage to be reserved.

1.3.1.2 Data management

In previous discussions it was indicated that anything that can be processed by a program or routine in the system is considered by the control programs as data. Thus a particular collection of characters and symbols which are manipulated by one program may very well constitute another program, which in turn operates on other data. This flexibility is made possible by the use of an organizational concept known as a *data set*, a collection of information presented to the system as an entity. As such it has a unique name which refers to that collection and is independent of any device on which it may be stored or any form in which it may be organized. These attributes are defined separately and may therefore be changed without affecting the integrity of the data set name. It is this independence which allows the programmer to refer to a particular data set name (which he uses to identify a data file) in his program and change the device assignment externally as often as is necessary.

The data-management programs, acting on specifications supplied by the programmer and additional information generated by other control programs, create data sets from collections of data, allocate auxiliary storage for them (on tape or disk), and keep track of the usage of such storage. In addition, they contain extremely powerful mechanisms which allow any number of data sets to be organized into a library whose components can be referenced at any time. The key to such a library is the *catalog* which is kept by the data-management programs and contains a unique name for each data set, together with information indicating where it is stored. Consequently a reference to its name is the only specification required to make use of that data set. The data-management programs, acting on that reference, will search the catalog and locate the data set. As part of the process of creating and storing the data set, these programs augment the actual data with a complete description of it, so that once the appropriate location is determined and accessed, sufficient information is immediately available to the data-management programs for them to determine the organization of the data and properly handle its transmission to and from the processor.

Data sets consist of a series of *records,* where each record represents some logically related collection of characters, symbols or values. For example, if the data set happens to be a program, each record would consist of an instruction. If the data set were a payroll file, each record would represent the information pertaining to one employee. To make more efficient use of auxiliary storage, several of these *logical records* are often strung together (*blocked*) to form a composite which is handled by the operating system

as a single large record (*physical record*). When a data set is being created, the data-management programs will perform such blocking operations automatically, based on programmer specifications. When a previously generated data set is being used by a program, its description (obtained either as a set of system instructions external to the program, automatically through the catalog, or as part of the program itself, in the case of COBOL) allows the data-management routines to handle all I/O transactions, including the separation of physical records into logical records (*deblocking*) to fit the specified needs of the processing program.

There are a large number of additional operations handled by the data-management programs, including protection of data sets (so that access to them is limited to certain users), and retrieval of individual records within a data set. Complete descriptions of the operations performed by the various types of available data-management programs are given in the appropriate manuals.*

1.3.1.3 Other Control Programs

In addition to the basic types of operations outlined above, there are a number of facets of the control functions which are intended for special types of configurations. For example, there is an entire array of routines for handling operations with graphical data via the 2250 and 2260 display devices. Similarly, a complete repertoire of control programs can be incorporated when a facility is to be equipped for message switching (exchange of information between terminals), remote inquiry, and other operations connected with teleprocessing. Those wishing to pursue these areas can find pertinent references in the bibliography of System/360 manuals.[43]

1.3.2 Processing Programs

System/360 software is structured to allow the incorporation of an indefinite number of processing programs and routines. Although all programs, when viewed in a broad sense, perform some type of processing, it is convenient to consider such programs as compilers, assemblers, utilities and applications programs as a separate group. The basic difference between these and other operating system components lies in the fact that processing programs are not concerned with controlling the scheduling and execution of other programs.

Processing programs can be divided functionally into compilers and assemblers (or language translators), service or utility programs, and applications programs. We shall address ourselves to the first two of these categories, since

*For Tape Operating System, see Ref. 38; for Disk Operating System, Refs. 39 and 40; for Operating System, Refs. 41 and 42.

the programs therein are oriented toward the general population of users, while the spectrum of applications programs cannot be anticipated. It should be understood that this categorization represents a conceptual vehicle for convenience of discussion. So far as the system is concerned, all processing programs are handled in the same way and have the same organizational significance.

1.3.2.1 Language Translators

The System/360 programmer may write his programs in a wider variety of source languages than was available to him with previous systems. Some of these languages (such as COBOL and FORTRAN) existed before and have been enlarged to incorporate elements which exploit the system's capabilities more fully. Others, like the macroassembler and PL/I, are more recent in origin and correspond more directly to third-generation concepts. Each of these languages is designed to facilitate the development of programs for some given broad area of applications.

1. *Languages for commercial and business applications.* Translators for the widely used COBOL language and a tabular language called Report Program Generator (RPG) are available in various versions for the various System/360 models. These versions differ with respect to the number of language features included in them and their usage depends on the type of hardware/software configuration available. A detailed discussion of COBOL is given later in this book. Information on RPG can be found in Refs. 44, 45, 46, and 47.
2. *Languages for scientific applications.* Although the distinction between "business" and "scientific" types of programs is becoming less as information-processing technology develops, certain high-level languages are still considered to be specifically intended for scientists and engineers. Two of the most common of these are ALGOL and FORTRAN. System/360 processing programs include translators for each of these, and both languages will be discussed in subsequent sections of this book.
3. *Universal Languages.* In an effort to provide a common vehicle for all types of programmers, a new language called PL/I (Programming Language/One) has been designed and implemented on several types of third generation computing systems, with System/360 being preeminent. In addition to incorporating many of the advantageous features of FORTRAN and COBOL, this language also provides the high-level language programmer with access to many system features and capabilities which would otherwise compel him to use a machine-oriented language.

   Included also are several versions of the basic type of universal source language, i.e., a symbolic assembly language. Although each instruction in this type of language is not as powerful as in a high-level language,

the various assembly language commands correspond most closely to machine operations. As a result, programs written in this language tend to be more efficient (both with respect to storage use and execution time) because the programmer has more direct control over the individual operations. For certain processor models, the basic language is supplemented by a series of macros which allow the programmer to specify a predefined sequence of instructions with a single command. When the assembly language translator produces the corresponding object program, that sequence will actually appear in the program in place of the macro instruction which referred to them.

4. *Other Languages.* A large number of additional language translators have been added to various versions of System/360 software. These give the programmer access to several list processing languages such as SNOBOL[48] and LISP,[49] algebraically oriented teaching languages such as IITRAN[50] and simulation languages such as SIMSCRIPT.[51] Although these and other translators are not furnished as part of the software package supplied by IBM, it is usually possible to obtain them.

1.3.2.2 Utility Programs

Certain types of manipulations find common use in a wide range of information processing applications. While not predominantly computational in nature, they are to a large extent sufficiently intricate to require lengthy sequences of instructions. System/360 software includes a number of such routines among its processing programs, together with the necessary mechanisms to allow the programmer to incorporate them as steps in his job. Since they are generally concerned with data arrangements, transmission and similar preparatory operations, they are commonly termed service or utility programs.

1. *The sort/merge program.* This program, operating under supervision of the control programs, rearranges the order of records in a data set. The new sequence is dictated by some criterion contained within each record, as specified by the programmer. Since the size and number of records involved in such an operation many prohibit the internal storage of the entire data set at one time, it is often necessary to divide the sorting operation into a number of phases in which only some of the records are resequenced. Each of these phases must be followed by another operation in which several of these groups of sorted records are merged to form a smaller number of groups, with each group consisting of a larger number of sorted records. The sort/merge program is sufficiently versatile to design the processing for each set of circumstances so that effective use is made of all available input/output units.[52-56]

2. *Other data-arrangement programs.* A number of additional utility programs are available for transferring data sets from one I/O device to another. This is often done without a specific request by the programmer, based on instructions from the control program. For example, a set of results produced by the user's program and intended for display on the line printer may first be directed (without his knowledge) to a direct-access device while the printer is busy, to be transferred from there at some more propitious time. Facilities are also available to effect such transmission with additional editing.

   Other programs may be called upon for modifying data sets which have previously been added to the system library. Corresponding changes in the catalog can also be made to reflect these modifications. In support of these activities, another utility program can be used to produce a detailed listing of the entire library at any time. Utility programs are described further in Refs. 57, 58, 59, and 60.

1.3.2.3 The Linkage Editor

Although classified as one of the service or utility programs, the linkage editor warrants special attention because its function provides special insight into one of System/360's most important operational characteristics.

In many earlier systems the processing of the source language program by an assembler or a compiler/assembler sequence produced a machine language object program in executable form. Specifically, each instruction and data item was assigned a definite memory location (absolute address) so that subsequent loading of the program was a straightforward operation. Thus every time an object program was brought into memory and executed, it occupied the same set of locations.

This method of processing cannot be employed in System/360 without severely limiting the accessibility of many if not most of its operating features. In order for the control program to be able to set up and maintain protected storage areas, to manage multiprogramming systems and to bring programs and data in and out of memory at various times, a mechanism must be available whereby the storage locations of program instructions are not assigned until that program is actually loaded immediately prior to execution.

Consequently, a 360 object program (i.e., the product of compilation or assembly) is not directly executable. Instead, each instruction and data item is assigned an address which is not absolute but relative to some reference value that is not established until the program is to be loaded. Such a program is said to be *relocatable.* The process of assigning the actual locations to such a program immediately prior to its loading and execution is handled by the linkage editor. Since this program is part of the overall software structure, it receives current information regarding the availability

of main storage, thus allowing it to make the appropriate assignment. In this manner the portion of memory from which a program is executed can change from run to run, depending on conditions that prevail in the system.

The ability to delay the assignment of absolute location to programs until immediately before execution time provides several additional benefits, including the following. It is possible to use the linkage editor to combine several routines, in object form, into a large program. This avoids the necessity of having to recompile or reassemble the respective source programs. In addition, the linkage editor can be used to divide programs which are too large to be accommodated in memory. By handling the location assignment automatically, the linkage editor is able to make arrangements so that latter portions of such a program are held temporarily in external storage until part of the program has executed. When that happens the externally stored portions are assigned to the locations of the instructions already processed, so that the new ones can be brought in to replace the others. (This process is known as overlaying.)

A detailed description of the linkage editor and its function can be found in Ref. 61.

1.3.3 Job Control Language

One of the most powerful aspects of System/360 software is the extensive vehicle provided for communication between users and the job scheduling components of the operating system. The instructions provided for this purpose are known collectively as the Job Control Language (JCL). These instructions are somewhat analogous to the control cards used to describe a particular processing run in some of the earlier systems. However, JCL represents a much more versatile control instrument than its predecessors. First of all, its organization is such that the instructions, operands, and options contained in its repertoire constitute a language in the same sense that FORTRAN or COBOL is a language. By writing a sequence of simply structured commands in this language the System/360 programmer or user can provide an entire set of specifications for a job at the time it is to be executed. Information transmitted in JCL instructions can include requirements for programs to be used in each step, descriptions of data sets, and definitions of which devices to use. In addition, there are JCL instructions to allocate and organize storage on external devices, to access the catalog of stored data sets, to access procedures for making a data set available ("known") to several steps in a given job, and to permit the sharing of a data set by more than one job in a multiprogramming environment.

JCL consists of six basic types of instructions (usually called *control statements*), with a multitude of options and parameters available for the various statement types. At this point we shall merely enumerate the types, deferring more detailed considerations to a later point.

1. *The* JOB *statement* identifies the beginning of a new job. When a sequence of several jobs, each preceded by its respective JOB statement, is submitted to the operating system, this is known as a *job stream.* Some of the options in this statement allow the user to include information for accounting routines, requests for certain displays produced by the job scheduler, priority designations and conditions for abnormal job termination.
2. *The* EXEC (*execute*) *statement* indicates the beginning of a job step and defines the program to be used in performing this step. Options are provided to introduce job-step accounting information, to set up criteria for bypassing the entire step, to assign an execution time limit and to transmit information directly to a processing program.
3. *The* DD (*data definition*) *statement* provides a complete description of an existent data set, or of one to be created. In addition to defining the organization of the data themselves, the statement requests the allocation of one or more peripheral devices and provides labeling information. It is this particular statement which is changed just prior to execution time whenever a processing program is to be used with data coming from or going to a different device than was employed during a previous run.
4. *The command statement* provides the vehicle for the computer operator to enter information into the input stream. With this statement the operator can request certain displays and assign or deactivate system input or output units.
5. *The delimiter statement* is used as a punctuation mark in the input stream when data are entered as part of the job step. This statement serves to distinguish the data from other control statements. Thus, a source program (which represents data to a compiler) is followed by a delimiter statement.
6. *The null statement* is used in special types of jobs to define the end of the control statements.

PROBLEMS

1. Define the term "architecture" as used in System/360 context.
2. Describe the basic procedures involved in overlapped operation and state the benefits of such operations.
3. What is a "job" in System/360 terminology and how does it differ from the jobs handled by many earlier systems?
4. Differentiate between time sharing, multiprocessing, and multiprogramming.
5. What is meant by the term "upward compatibility"?
6. How do the operating characteristics of selector and multiplexer channels differ?

7. Give a conceptual description of data transmission to or from a System/360 central processor.
8. What is the difference between an I/O device, an I/O channel, and an I/O control unit?
9. Describe three ways of setting up a 360 system with more than one central processor.
10. What is bulk core? How does it differ from "regular" core?
11. How large is a 16K memory? How large is H size?
12. What is meant by memory protection?
13. Distinguish between sequential and direct access.
14. Describe the properties of a job with respect to System/360 software.
15. Differentiate between a job step and a task.
16. Define a data set. How does it differ from a data file?
17. What is a relocatable program? Why are System/360 programs constructed in this manner?
18. Describe the functions of Job Control Language.

REFERENCES

1. Amdahl, G. M., G. A. Blaauw, and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal of Research and Development* (April 1964).
2. *IBM System/360 System Summary*, Form A22-6810. IBM Corp., White Plains, N.Y.
3. *IBM 2361 Core Storage—Original Equipment Manufacturer's Information*, Form A22-6869. IBM Corp., White Plains, N.Y.
4. *1442—Component Description and Operating Procedures*, Form A21-9025. IBM Corp., White Plains, N.Y.
5. *2501—Component Description and Operating Procedures*, Form A21-9020. IBM Corp., White Plains, N.Y.
6. *2520—Component Description and Operating Procedures*, Form A21-9027. IBM Corp., White Plains, N.Y.
7. *2540—Component Description and Operating Procedures*, Form A21-9033. IBM Corp., White Plains, N.Y.
8. *1403 Printer—Component Description*, Form A24-3073. IBM Corp., White Plains, N.Y.
9. *1404 Printer*, Form A24-1446. IBM Corp., White Plains, N.Y.
10. *1443 and 1445 Printers*, Form A24-3120. IBM Corp., White Plains, N.Y.
11. *2250 Display Unit Model 1—Component Description*, Form A27-2701. IBM Corp., White Plains, N.Y.
12. *2250 Display Unit Model 3—Component Description*, Form A27-2721. IBM Corp., White Plains, N.Y.
13. *2260 Display Station—Component Description*, Form A27-2700. IBM Corp., White Plains, N.Y.
14. *2400—Component Description*, Form A22-6866. IBM Corp., White Plains, N.Y.

15. *2400 Magnetic Tape Units—Original Equipment Manufacturers' Information*, Form A22-6862. IBM Corp., White Plains, N.Y.
16. *7340 Hypertape Drive Model 3 with System/360*, Form A22-6828. IBM Corp., White Plains, N.Y.
17. *7340 Hypertape Drive—Original Equipment Manufacturers' Information*, Form A22-6861. IBM Corp., White Plains, N.Y.
18. *Introduction to System/360 Direct Access Storage Devices and Organization Methods*, Form C20-1649. IBM Corp., White Plains, N.Y.
19. *2311 Disk Storage Drive—Original Equipment Manufacturers' Information*, Form A26-3567. IBM Corp., White Plains, N.Y.
20. *2314 Direct Access Storage Facility—Component Description*, Form A26-3599. IBM Corp., White Plains, N.Y.
21. *2303 Drum Storage—Original Equipment Manufacturers' Information*, Form A26-5663. IBM Corp., White Plains, N.Y.
22. *2301 Drum Storage—Component Description*, Form A22-6895. IBM Corp., White Plains, N.Y.
23. *2301 Drum Storage —Original Equipment Manufacturers' Information*, Form A26-5661. IBM Corp., White Plains, N.Y.
24. *2321 Data Cell Drive—Original Equipment Manufacturers' Information*, Form A26-3574. IBM Corp., White Plains, N.Y.
25. *Data Cell Handling Guide*, Form A26-3633. IBM Corp., White Plains, N.Y.
26. *2302 Disk Storage—Component Description*, Form A26-5988. IBM Corp., White Plains, N.Y.
27. *Data Communications Configurator*, Form A22-6824. IBM Corp., White Plains, N.Y.
28. *Basic Programming Support—Basic Tape System Programmer's Guide*, Form C24-3354. IBM Corp., White Plains, N.Y.
29. *Basic Programming Support—Programming Systems Summary*, Form C24-3420. IBM Corp., White Plains, N.Y.
30. *Basic Operating System—Programmer's Guide*, Form C24-3372. IBM Corp., White Plains, N.Y.
31. *Basic Operating System Operating Guide—Control Programs and Assembler*, Form C24-3450. IBM Corp., White Plains, N.Y.
32. *Tape Operating System—Concepts and Facilities*, Form C24-5030. IBM Corp., White Plains, N.Y.
33. *Tape Operating System—Operating Guide*, Form C24-5021. IBM Corp., White Plains, N.Y.
34. *Disk Operating System—Operating Guide*, Form C24-5022. IBM Corp. White Plains, N.Y.
35. *Operating System Introduction*, Form C28-6534. IBM Corp., White Plains, N.Y.
36. *Operating System Concepts and Facilities*, Form C28-6536. IBM Corp., White Plains, N.Y.
37. *Operating System—System Programmer's Guide*, Form C28-6550. IBM Corp., White Plains, N.Y.
38. *Tape Operating System—Extended Supervisor and I/O Macros*, Form C24-5035. IBM Corp., White Plains, N.Y.

39. *Disk Operating System—Data Management Concepts*, Form C24-3427. IBM Corp., White Plains, N.Y.
40. *Disk Operating System—Extended Supervisor and I/O Macros*, Form C24-5037. IBM Corp., White Plains, N.Y.
41. *Operating System—Supervisor and Data Management Services*, Form C28-6696. IBM Corp., White Plains, N.Y.
42. *Operating System—Supervisor and Data Management Macro-Instructions*, Form C28-6647. IBM Corp., White Plains, N.Y.
43. *IBM System/360 Bibliography*, Form A22-6822. IBM Corp., White Plains, N.Y.
44. *Basic Programming Support—Report Program Generator Language Specifications*, Form C24-3418. IBM Corp., White Plains, N.Y.
45. *Basic Operating System—Report Program Generator Language Specifications*, Form C24-3387. IBM Corp., White Plains, N.Y.
46. *Disk/Tape Operating System—Report Program Generator*, Form C26-3570. IBM Corp., White Plains, N.Y.
47. *Operating System—Report Program Generator Language*, Form C24-3337. IBM Corp., White Plains, N.Y.
48. Goff, A. G., "Operating System SNOBOL," *Brown University Computing Review* (Special Supplement), Vol. 1, No. 2 (June 1967).
49. McCarthy, J., P. N. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, *LISP 1.5 Programmer's Manual*. Cambridge, Mass.: M.I.T. Press, 1965.
50. Bauer, C. R., A. P. Peluso, and W. S. Worley, Jr., *IITRAN/360* Reading, Mass.: Addison Wesley Publishing Co., Inc., 1967.
51. Dimsdale, B. and H. M. Markovitz, "A description of the SIMSCRIPT Language," *IBM Systems Journal*, Vol. 1 (1964).
52. *Basic Programming Support—Sort/Merge Programs—Specifications*, Form C24-3320. IBM Corp., White Plains, N.Y.
53. *Basic Operating System—Sort/Merge Program*, Form C24-3321. IBM Corp., White Plains, N.Y.
54. *Tape Operating System—Sort/Merge Program Specifications*, Form C24-3438. IBM Corp., White Plains, N.Y.
55. *Disk Operating System—Sort/Merge Program Specifications*, Form C24-3444. IBM Corp., White Plains, N.Y.
56. *Operating System—Sort/Merge Program*, Form C28-6543. IBM Corp., White Plains, N.Y.
57. *Basic Programming Support—Basic Utility Program Specifications*, Form C28-6505. IBM Corp., White Plains, N.Y.
58. *Basic Operating System—Utility Program Specifications*, Form C24-3409. IBM Corp., White Plains, N.Y.
59. *Tape/Disk Operating System—Utility Program Specifications*, Form C24-3465. IBM Corp., White Plains, N.Y.
60. *Operating System—Utilities*, Form C28-6586. IBM Corp., White Plains, N.Y.
61. *Operating System—Linkage Editor*, Form C28-6538. IBM Corp., White Plains, N.Y.

Chapter 2

System/360 Data Handling

One of the most important aspects of System/360's design is the emphasis on versatility. In the previous chapter some general instances of this were seen with respect to the organization of the hardware (e.g., standard I/O interfaces) and software (such as the ability to change I/O device designations without affecting the applications program). In this chapter we shall examine a crucial aspect of this versatility by taking a closer look at how various types of data are stored and organized in System/360 processors.

2.1 THE SYSTEM/360 ALPHABET

Until the 360 series of computers was introduced, most machines were designed to store letters, digits, and other individual characters as prescribed combinations of six binary digits. Use of this six-bit byte provided those systems with an "alphabet" of 64 possible characters. Expanded data requirements, together with a wider variety of data generating and transmitting equipment, have prompted the adoption of an 8-bit byte as the basic representational storage unit in System/360 processors. This raises the size of the acceptable "alphabet" to 256 different types of characters.

The code developed for the 8-bit byte in System/360 is known as the

TABLE 2.1 INTERNAL REPRESENTATION OF EBCDIC CHARACTERS

0 1 2 3	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1	1 0 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1
0 0 0 0		CC1*	** →		CC2*	CC3*	CC4*	CC5*	** →							
0 0 0 1		** →		CC6*	CC7*	CC8*	CC9*	CC10*	** →		CC11*	** →				
0 0 1 0		CC12*	CC13*	CC14*	**	CC15*	CC16*	CC17*	CC18*	** →		CC19*	** →			
0 0 1 1		** →				CC20*	CC21*	CC22*	CC23*	** →						
0 1 0 0		blank	** →										]or[	.	<	(	+	]or[
0 1 0 1		&	** →										]or[	$	*	)	;	or
0 1 1 0				** →										.	%	_	>	?
0 1 1 1		** →												=	@	'		**
1 0 0 0		**	a	b	c	d	e	f	g	h	i	**	{	≤	("	+"	±	
1 0 0 1		**	j	k	l	m	n	o	p	q	r	**	~	□	)"	÷		
1 0 1 0		**	~	s	t	u	v	w	x	y	z	**	—	⊏	[	¬	^	
1 0 1 1		0"	1"	2"	3"	4"	5"	6"	7"	8"	9"	**	⊐	⊐	]	¬	—	
1 1 0 0		**	A	B	C	D	E	F	G	H	I	** →						
1 1 0 1		**	J	K	L	M	N	O	P	Q	R	** →						
1 1 1 0		** →		S	T	U	V	W	X	Y	Z	** →						
1 1 1 1		0	1	2	3	4	5	6	7	8	9	** →						

**represents an unassigned bit pattern
bit strings indicated by CCn are used for internal control functions
"these characters are superscripts.
Where more than one symbol is shown for a particular bit pattern, the one printed depends on the type of print chain used.

Extended Binary Coded Decimal Interchange Code (EBCDIC). Tables 2.1 and 2.2 give the bit patterns for the various characters, together with the corresponding keypunch patterns. Note that graphical equivalents do not exist for all of the characters. Some have internal control functions and others are still unassigned. This does not prevent the programmer from using these bit combinations in his input and testing for them as his purposes dictate.

The code is called "extended" because it reflects an effort to maintain maximum compatibility with the binary coded decimal (BCD) code used in earlier systems. This is done by assigning bit combinations such that a 6-bit subset of a character's EBCDIC representation matches the BCD code for that character. For example, the BCD code for a W(100110) is reflected in the corresponding EBCDIC code (11100110), thus helping make it possible for data recorded on cards and tapes with earlier equipment to be read by System/360 hardware. This type of consistency extends to other charac-

TABLE 2.2 KEYPUNCH PATTERNS FOR EBCDIC CHARACTERS

			1	2	3	4	5	6	7	8	9
		**	1	2	3	4	5	6	7	8	9
12		&	A	B	C	D	E	F	G	H	I
12	11	**	j	k	l	m	n	o	p	q	r
12	0	**	a	b	c	d	e	f	g	h	i
12	9	1	** →			CC2*	CC3*	CC4*	CC5*	**	
12	8	H	**	ç	·	<	(	+″	1	**	**
12	11 0	**	1″	2″	3″	4″	5″	6″	7″	8″	9″
12	11 8	** →			}	%	)″	±	=		**
12	0 8	** →			{	≤	(″	+″	+		**
12	11 0 8	**	°″	**	¨	⊐	]	≠	–		**
12	0 9 8	**	CC1*	** →							
	11	–	J	K	L	M	N	O	P	Q	R
	11 0	**	¢	s	t	u	v	w	x	y	z
	11 9	R	** →		CC6*	CC7*	CC8*	CC9*	CC10*	**	
	11 8	** →	!	S	*	)	;	¬			**
	11 9 8	** →	CC11*	** →							
	11 0 8	** →		⌙	⊏	[	≥	·			**
	11 0 9 8	**	CC12*	** →							
	0	0	/	S	T	U	V	W	X	Y	Z
	0 9	Z	CC13*	CC14*	**	CC15*	CC16*	CC17*	CC18*	**	
	0 8	** →		,	**	–	>	?			**
	0 9 8	** →	CC19*	** →							
	9	9	** →			CC20*	CC21*	CC22*	CC23*	**	
	8	8	**	:	=	§	·	=	″		**

ters in the code. Examination of Table 2.1 indicates, for example, that representations for uppercase and lowercase letters differ only by one bit, thereby facilitating comparisons and other internal operations.

Another code has been developed for representing characters as 8-bit bytes. This is known as the American Standard Code for Information Interchange (ASCII) and finds frequent use in data transmission. The code is shown in Table 2.3. Although System/360 processors are designed to accept data recorded in ASCII code, the internal circuits are oriented toward EBCDIC representation and will perform the necessary conversions. Consequently, our discussion will lean in that direction.*

2.2 ORGANIZATION OF MEMORY

As indicated in previous sections, System/360 primary memory is organized as a series of contiguous 8-bit bytes. (There is a ninth (parity) bit for internal checking, but the programmer is not usually concerned with this). Each byte has a separate address, with the first one being zero.

In order to support the variety of information-handling activities for which System/360 is intended, it is insufficient to restrict memory organization to single bytes. Consequently, there are several ways in which primary storage can be treated.

2.2.1 Basic Memory Divisions

When a portion of memory is reserved for a particular data item, that portion is referred to as a *field.* The length of a field (number of contiguous addressable storage elements) reserved for such a purpose usually depended on the type of processor being programmed and, somewhat less strongly, on the particular machine instruction being used. The memories of most so-called "scientific" computers were organized into addressable units (words) consisting of several bytes, and the majority of their internal computational and manipulative instructions were designed to handle individual words and could not directly manipulate part of a word or a group of words. On the other hand, computers that were marketed as "business" systems had memories that were organized so that their elements could be grouped into fields of any desired length, with the number of locations being referenced by a given instruction varying according to the way in which the memory was allocated in that program. Such machines, commonly known as variable word-length machines, provided this versatility at some penalty in internal processing efficiency.

System/360 combines the organizational features of fixed and variable word length machines by providing parallel sets of instructions for handling

*Because the ASCII is used in a large number of government data handling applications, it may be that future 360 processors will use this code for internal representation.

TABLE 2.3 INTERNAL REPRESENTATION OF ASCII CHARACTERS

	4 5 6 7	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1	1 0 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1
1 2 3																	
0 0 0		CC1*	CC2*	CC3*	CC4*	CC5*	CC6*	CC7*	CC8*	CC9*	CC10*	CC11*	CC12*	CC13*	CC14*	CC15*	CC16*
0 0 1		CC17*	CC18*	CC19*	CC20*	CC21*	CC22*	CC23*	CC24*	CC25*	CC26*	CC27*	CC28*	CC29*	CC30*	CC31*	CC32*
0 1 0		**	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**
0 1 1		**	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**
1 0 0		blank	!	"	=	$	%	&	'	(	)	*	+	,	−	.	/
1 0 1		0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
1 1 0		**	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**
1 1 1		**	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**
0 0 0		**	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**
0 0 1		**	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**
0 1 0		@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0 1 1		P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
1 0 0		**	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**
1 0 1		**	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**
1 1 0		**	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
1 1 1		p	q	r	s	t	u	v	w	x	y	z	**	**	**	CC33*	CC34*

** Represents an unassigned bit pattern.

* Bit strings indicated by CCn* are used for internal control functions.

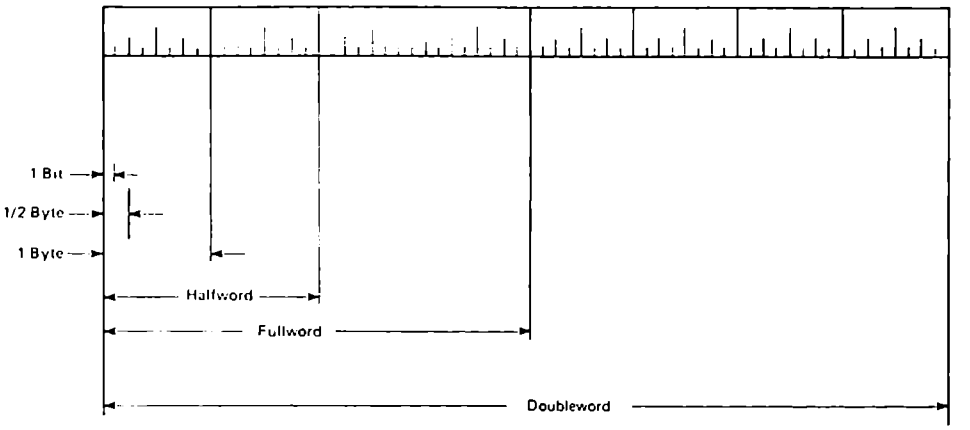


Figure 2.1 Organization of System/360 Memory

a variety of field sizes. That is, in addition to a set of capabilities for working with variable field lengths, the programmer has the opportunity of using certain instructions designed for fixed word lengths of several sizes.

The shortest field length, of course, is the byte, since each single byte has its own unique address. Correspondingly, there is a set of instructions for manipulating single bytes. By using an alternate set of instructions it is possible to manipulate two contiguous bytes at a time. This fixed length field is known as a *half-word.* A string of 4 consecutive bytes constitutes a *fullword* and appropriate instructions exist for manipulating that type of memory unit. In certain instances, it is also possible to refer to 8 consecutive bytes as a single field. This is known as the *doubleword.* Finally, the programmer can use a set of instructions in which he specifies the length of the field (number of bytes) to be manipulated. Field lengths ranging from 1 to 256 bytes can be processed in this way. This organization of memory is schematically depicted in Figure 2.1 and the corresponding arrangement of Assembler Language instructions to handle these various fields is discussed in the next chapter.

There are, of course, ample capabilities for handling data at the bit level. However, such manipulations are not directly related to the organization of memory because single bits are not addressable as such. Instead, they must be designated as specific subsets of particular bytes.

2.2.2 Addressing of Storage

When we spoke about the ability to execute System/360 programs from various locations in memory (Chapter 1) it was pointed out that the generation of relocatable object programs was crucial for this purpose. In order to understand this better we shall examine the basic addressing mechanism which makes this possible. In addition, we shall discuss related areas dealing with convenient methods for expressing internal locations.

2.2.2.1
Construction of Internal Addresses

Although there are various instruction formats to handle the different field lengths, they all share the common necessity of specifying an operand which must eventually be expressed as an internal address. Since System/360 design allows ultimate extension of primary memory to 16,777,216 (2^{24}) bytes, each internal address must be represented by a 24-bit binary number. It is System/360's indirect method of expressing such addresses, and the reasons behind it, that concern us at this point.

System/360 processors are equipped with 16 special memory units known as *general-purpose registers.* These are each 32 bits long and are numbered 0–15. They may be used for a variety of purposes, one of which is to store address information about operands. When used for that purpose a register supplies part of the information about an address to the logical circuitry, which combines it with data from the particular instruction being processed to obtain the complete address. Specifically, the low-order (rightmost) 24 bits of the register contain a binary number known as the *base address.* A portion of the instruction called the *base* specifies which of the registers (called the *base register*) is to be referenced for the base address and another part of the instruction, termed the *displacement,* indicates the location of the desired byte relative to the base address. Thus when the instruction is executed, the control circuits consult the base portion to find the proper register. The appropriate base address is then added to the quantity in the displacement portion of the instruction to obtain the required address.

As a simple example, let us suppose that the instruction format looks schematically as follows:

Operation	Base	Displacement

and a particular instruction contains

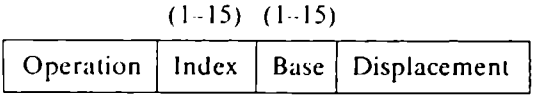
	Base (4)	Displacement (140)
Operation	0100	000010001100

Let us suppose further that general-purpose register number 4 contains a value in the rightmost 24 bits whose decimal equivalent is 3200:

	3200
00000000	000000000000110010000000

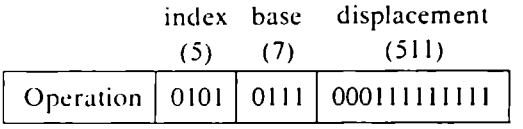
When this instruction is processed, the circuitry will consult register number 4, add 140 to 3200 and use 3340 as the operand's address. Register 0 cannot be used in this way for reasons which will be given later.

When certain types of instructions are used, it is possible to specify a second register (known as an *index register*) as part of the instruction which will then be used as one of the addends in calculating a final address. For these situations we can visualize the process as shown:



Total displacement = Index Register contents and Displacement
Effective address = Contents of register whose no. is specified in Base and Total Displacement

Thus the effective address will be calculated as the sum of the contents of the base and index registers, added to the value in the displacement portion of the instruction. For example, if register 5 contained 460 and register 7 contained 300, the instruction



would produce a final operand address of 300 + 511 + 460 or 1271.

Now we can take a look at some of the basic benefits realized from this addressing technique. As a start let us analyze the instruction components. Twelve bits are allotted for the displacement portion, thus permitting the programmer to specify a displacement of as much as 2^{12} or 4,096 locations from the base address. Since the base portion of the instruction can only specify a register number as high as 15, no more than 4 bits are required for that portion. Thus, as far as each individual instruction is concerned, only 12 + 4 or 16 bits are needed to provide a complete address specification, as opposed to the 24 bits that would be used if addressing were direct.

This 8-bit saving in instruction length results in considerable economy in processing time and storage use. Furthermore, the conceptual operation of the relocating mechanism can now be clearly seen. With this addressing method an entire program can be shifted from one area of storage to another merely by changing the base address.

Much of the clerical work involved in assigning and reassigning values to registers and calculating addresses from base and displacement values is done automatically by system software. Such housekeeping includes procedures for storing base values in ordinary memory to free registers for other uses. At the proper time these values are replaced so they are in the register when the user's program expects them to be there. Such transfers of information to and from registers are very rapid, so that effective operations can be maintained with only 16 registers.

2.2.2.2
Expression of Addresses—Hexadecimal Notation

Once a particular address has been calculated, the resulting 24-bit binary string constitutes a reference to a specific byte in memory. Depending on the type of instruction in which this reference is made, the address may pertain to a single byte or a larger field. When the latter is the case, the address refers to the first (i.e., leftmost) byte of that field.

The representation of internal addresses as 24-bit binary numbers is convenient for the processor but very awkward for the user. On the other hand, expression of these locations in decimal form for display purposes constitutes a sizable inconvenience in terms of software design. Consequently it was decided to use a condensed notation consisting of a number system based on 16. This is known as the *hexadecimal system* and is used by various software routines to display addresses and other information pertaining to internal processing activities.

The choice of 16 as a base necessitates the definition of a set of symbols

TABLE 2.4 EXAMPLES OF HEXADECIMAL NUMBERS AND THEIR DECIMAL EQUIVALENTS

Decimal Value	Hexadecimal Value
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
20	14
31	1F
50	32
64	40
100	64
128	80
255	FF
500	1F4
1000	3E8
4096	1000

such that each of the magnitudes 0–15 can be expressed with single characters. For this purpose it was decided to use 0–9 in their usual context and extend the sequence with A representing 10, B = 11, C = 12, etc. through F = 15. Table 2.4 gives a somewhat longer sequence of decimal numbers and their hexadecimal equivalents so that the student can orient himself more easily. Note that positional consistency is maintained. Just as

$$164_{10} = 1 \times 10^2 + 6 \times 10^1 + 4 \times 10^0$$

we can say that

$$164_{16} = 1 \times 16^2 + 6 \times 16^1 + 4 \times 16^0.$$

Similarly,

$$2F3 = 2 \times 16^2 + F \times 16^1 + 3 \times 16^0$$

or, in decimal terms,

$$\begin{aligned} 2F3 &= 2 \times 256_{10} + 15_{10} \times 16_{10} + 3 \times 1 \\ &= 512_{10} + 240_{10} + 3 = 755_{10}. \end{aligned}$$

The choice of 16 as a base is prompted by the fact that $16 = 2^4$, so that each group of 4 bits (or a half byte) can be represented by a single hexadecimal digit. Hence a 24-bit address can be written as a 6-digit hexadecimal number. We can see how this works by looking at the binary address:

001101000110110001001011

Expression of this number in hexadecimal form merely involves partitioning of the string into groups of 4 bits and replacing each group with the corresponding hexadecimal digit. Thus

0011	0100	0110	1100	0100	1011
3	4	6	C	4	B

$346C4B_{16}$ is equivalent to $001101000110110001001011_2$.

We shall see later on that this type of notation is useful in displaying address contents and instruction codes as well as the addresses themselves. Also, we shall find that certain arithmetic operations are actually performed internally via the hexadecimal system.

2.3 INTERNAL REPRESENTATION OF NUMBERS

We have seen that System/360 processors are designed to accept a wide variety of numbers, letters, and other characters and store them one character per byte in accordance with the EBCDIC code. This is a fundamental property of the system and is not directly related to its computational functions. That is, all input is treated as strings of bytes representing characters, regardless of the contents of that input or the user's intent. Thus the

character string AC12Z, which might represent a part number or other identifier, would be stored as

A	C	1	2	Z
11000001	11000011	11110001	11110010	11101001

By the same token, an input value of −43.69, when recorded that way on a punched card or other input medium, will be stored as 6 separate characters, namely

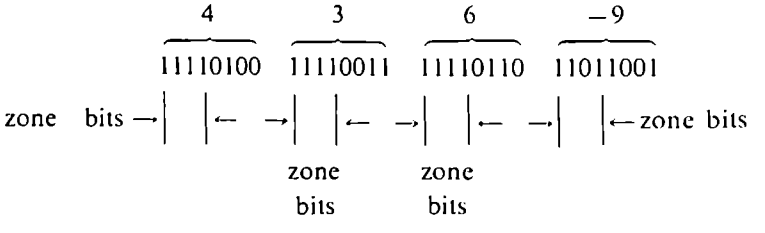
−	4	3	.	6	9
01100000	11110100	11110011	01001011	11110110	11111001

Note that there is no cognizance of the fact that the value of a particular digit is dictated by its position with respect to the decimal point, or that the minus sign refers to the entire value and not just the adjacent digit.

Because of the inability to distinguish between the meanings of numbers, letters, and other symbols, there are no instructions for performing arithmetic operations on numerical data stored in this form. Instead, various other forms are used and instructions are made available for all types of conversion.

2.3.1 Zoned-Decimal Format

The value −43.69, as used in the previous example, is an example of the least economical way of representing a number, both as input and in memory. Instead of requiring six columns on a punched card the same number could be made to occupy only four columns, one for each digit, by punching the sign in the same column as the low-order digit and omitting the decimal point. This, of course, is very common practice, with the programmer keeping track of the decimal point. When entered in this manner, (i.e., $436\bar{9}$) its representation in memory would be



This is known as the *zoned-decimal* format. In this format all except the low-order digit are represented according to the EBCDIC code. The decimal point itself is never explicitly indicated and the low-order byte contains a representation of the sign in the leftmost four bits (known as the *zone bits*). The EBCDIC sign conventions for zoned-decimal format are as follows. Unsigned numbers have 1111 in the zone bits of the low-order digit; numbers with an explicit plus sign contain 1100 in the zone bits of the low-order

TABLE 2.5 EXAMPLES OF NUMERICAL DATA IN ZONED DECIMAL FORMAT (EBCDIC)

Decimal Value	Hexadecimal Equivalent	Zoned Decimal Equivalent		
−31	−1F		11110011	11010001
629	275	11110110	11110010	11111001
+0	+0			11000000
72	48		11110111	11110010
0	0			11110000
−11	−B		11110001	11010001

digit, and 1101 is used for negative values. When the ASCII code is used, the conventions are 0101, 1010, and 1011 for unsigned, positive and negative numbers, respectively. Table 2.5 gives some additional examples.

Although the zoned-decimal format does provide some savings in storage use, it is still basically byte-oriented and therefore not suitable for arithmetic manipulations. However, it is more consistent with standard conventions for input.

2.3.2 Packed-Decimal Format

If we know in advance that a particular data item is going to be a number, we can economize further by using the *packed-decimal* format. The principle behind this type of storage can be developed by examining the zoned-decimal representation of the number used in the previous example ($436\overline{9}$). This is reproduced here for convenience:

$$\overbrace{11110100}^{4}\ \overbrace{11110011}^{3}\ \overbrace{11110110}^{6}\ \overbrace{1101}^{-}\overbrace{1001}^{9}$$

Note that the zone bits of all but the low-order digit contain 1's. Since these bits are necessary only to distinguish numbers from letters or other characters (see Table 2.1) they can be eliminated in this format without loss of information. In addition, it turns out to be more convenient to place the 4-bit sign indicator to the right of the low-order digit. The resulting packed decimal representation of the value $436\overline{9}$ is

$$\overbrace{0000}^{0}\ \overbrace{0100}^{4}\ \overbrace{0011}^{3}\ \overbrace{0110}^{6}\ \overbrace{1001}^{9}\ \overbrace{1101}^{-}$$

There are an extra four zeros to the left of the first digit. This surplus is necessary in order to make the value come out to an integral number of bytes. Thus the value $8436\overline{9}$ could be represented by the same amount of storage in packed-decimal format:

$$\overbrace{1000}^{8}\ \overbrace{0100}^{4}\ \overbrace{0011}^{3}\ \overbrace{0110}^{6}\ \overbrace{1001}^{9}\ \overbrace{1101}^{-}$$

TABLE 2.6 EXAMPLES OF NUMERICAL DATA STORAGE IN PACKED-DECIMAL FORMAT (EBCDIC)

Decimal Value	Hexadecimal Equivalent	Packed-Decimal Equivalent
−31	−1F	0000 0011 0001 1101
629	275	0110 0010 1001 1111
+0	+0	0000 1100
72	48	0000 0111 0010 1111
0	0	0000 1111
−11	−B	0000 0001 0001 1101

since the assignment of a half-byte for each digit and the sign comes out to an even number of half-bytes (an integral number of bytes). Table 2.6 shows packed-decimal representation for several additional examples.

The instructions which manipulate packed-decimal data can handle fields from 1–16 bytes in length. When the programmer is sure that a particular value will never fall below zero he can increase this range by an order of magnitude in either direction by using the extra half-byte (usually reserved for the sign) for an additional digit. (Further extensions are possible, of course, through scaling). As we shall see later on, the length of a packed-decimal field can be set by the programmer based on the anticipated range, thus giving him control over the use of memory for his numerical constants and variables. If a particular field is to be used merely for storage (as the destination for an input value) the length determination is straightforward, requiring merely the knowledge of the anticipated range, number of significant digits, and sign requirements. For example, if the programmer knows that values for a given variable will be expressed to the nearest hundredth and will range from −30.00 to 1154, he would set up a field of four bytes for that purpose (one half-byte for the sign, three bytes for the six significant digits (two to the right and four to the left of an assumed decimal point), and the remaining half-byte, which will always contain zeros, goes along for the ride).

If a packed-decimal field is to be set up for a value which represents the result of some calculations, the determination of its length may be somewhat more involved. If that result were to represent a sum, for example, sufficient room would have to be allowed to accommodate the anticipated number of carries. If a product or quotient is to be contained there, the programmer must be guided by the size of the individual components.

2.3.3 Binary Format

System/360 also provides a set of instructions for manipulating numbers in *binary format.* In contrast to packed-decimal instructions, which allow the data field length to vary from 1 to 16 bytes, the binary instructions operate on fixed-length fields. This speeds up processing somewhat because

the instructions can deal with the field as a unit, since they "know" its size and don't have to proceed on a byte-by-byte basis.

Two binary field lengths are available (with corresponding sets of instructions for each). One is a halfword (16 bits) and the other extends for a full 32-bit word. In either case the leftmost bit is used to represent the sign (0 for positive, 1 for negative) and the remainder of the field (either 15 or 31 bits) is used to represent the magnitude of the number.

2.3.3.1 Positive Numbers

The form for positive binary numbers is true binary notation. That is, after the leftmost bit in the high-order (leftmost) byte, which contains a 0 for the sign, each succeeding bit represents a decreasing power of two. For example, the decimal number +872, when stored as a binary halfword, would appear as

0|0000011011101000

Similarly, the same value stored in a full binary word would mean that 16 additional zeros would appear at the left:

0|00000000000000000000001101101000

In either case we can read this number by starting from the leftmost 1 and summing the appropriate powers of 2. Thus

$$1 \times 2^9 + 1 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = +872$$

2.3.3.2 Negative Numbers

Although negative binary numbers could be expressed in true binary notation, it turns out that internal processing of such quantities is much more efficient when they are stored in so-called *two's complement* notation. The two's complement of a binary integer is found by changing all of the 0's to 1's and the 1's to 0's, then adding 1 to the low-order bit. We can illustrate this by taking the value −872 and first looking at it in true binary notation. Afterwards we shall make the conversion indicated above. Remembering that 1 is used in the high-order bit for negative binary numbers, the true binary half-word representation of −872 would be

1000011011101000

Changing all the 1's to 0's and vice versa (except, of course, for the sign bit) and then adding 1 to the rightmost bit produces

```
  1111110010010111
+                1
------------------
  1111110010011000
```

The same value, stored in a binary fullword, would merely have eight more high-order 1's. The internal processing advantage of the two's complement notation derives from the fact that adding the two's complement of a binary number is equivalent to subtracting the number. We can illustrate this with a simple example involving the subtraction 3 − 8. +3 expressed as a binary halfword is, of course,

```
0000000000011
```

Next we shall take the true binary notation of −8 and express it as a halfword in two's complement notation. This comes out

```
  1000000000001000
   1111111111110111
+                 1
-------------------
  1111111111111000
```

Performing the binary addition produces

```
  0000000000000011     +3
+ 1111111111111000     −8 in two's complement notation
-------------------
  1111111111111011
```

which is the two's complement notation of −5, the correct result. Just to tie everything up, we shall convert that value to true binary notation:

```
  1111111111111011
-                 1
-------------------
  1111111111111010
  1000000000000101
```

Negative quantities are automatically stored in two's complement form as part of the conversion process. Table 2.7 contains some examples of values

TABLE 2.7 EXAMPLES OF NUMERICAL DATA STORAGE IN BINARY FORMAT

Decimal Value	Hexadecimal Equivalent	Binary Halfword Equivalent
−31	−1F	1111111111100001
629	275	0000001001110101
+0	+0	0000000000000000
72	48	0000000001001000
0	0	0000000000000000
−11	−B	1111111111110101
3212	C8C	0000110010001100
−511	−1FF	1111111000000001

stored in binary format. Note that there is no such thing as −0 in binary form.

In addition to speeding up processing, use of the binary format provides fairly effective utilization of storage space. For example, the largest number that can be stored in two bytes using the packed-decimal format is 999 (since we have to use a half-byte for the sign). The same number of bytes (a halfword), when used for binary format, can accommodate a number as large as $2^{15} - 1$ or 32767.

2.3.4 Floating-Point Format

A third set of System/360 instructions for numerical processing makes use of data stored in *floating-point format*. This type of representation is extremely useful for those types of applications which deal with very large or very small numbers. Expression of such numbers causes problems because it is often necessary to include a large number of zeros in order to indicate the magnitude properly. Thus, although the numbers 178,000,000,000 and .000000000281 have only three significant digits each, it requires a great deal of space to represent them correctly in ordinary notation. To circumvent this, a shorthand form, known as scientific notation, has been devised and is used widely in mathematics, science, and engineering. Basically, a number expressed in this form consists of a product of two components. The first of these, known as the *fraction*, consists of the significant digits expressed as the decimal fraction nearest 1. This is multiplied by the power of 10 necessary to adjust the magnitude of that fraction (called the *characteristic*). Thus 178,000,000,000 would be written as $.178 \times 10^9$ and the fraction .000000000281 would appear as $.281 \times 10^{-9}$.

System/360 floating-point format uses basically the same type of representation as described above. The most important difference is that the number base for the internal representation of floating-point values is 16 rather than 10. That is, each number is still expressed as a product, as explained before. However, the mantissa is stored as a hexadecimal (not a decimal) fraction and the characteristic (the power of the base by which the mantissa is multiplied) is a power of 16. Thus, 39_{10}, whose decimal floating-point equivalent would be

$$.39 \times 10^2$$

would be stored (conceptually) as the hexadecimal floating-point equivalent of

$$2 \times 16^1 + 7 \times 16^0 = 27_{16}$$

or

$$.27 \times 16^2.$$

Additional examples are shown in Table 2.8.

TABLE 2.8 EXAMPLES OF DECIMAL VALUES AND THEIR HEXADECIMAL FLOATING-POINT EQUIVALENTS

Decimal Value	Hexadecimal Equivalent	Hexadecimal Floating-Point Equivalent
31	1F	$.1F \times 16^2$
−629	−275	$-.275 \times 16^3$
72	48	$.48 \times 16^2$
−11	−B	$-.B \times 16$
3212	C8C	$.C8C \times 16^3$
511	1FF	$.1FF \times 16^3$
.100	.1999	$.1999 \times 16^0$
1000	3E8	$.3E8 \times 16^3$
.06	.0F5C	$.F5C \times 16^{-1}$
.022	.05A2	$.5A2 \times 16^{-1}$

As with binary format, this form of numerical representation uses two alternative fixed field lengths. The smaller is a fullword (32 bits) and the larger option provides a doubleword (64 bits). The fullword format uses the first (leftmost) bit to specify the sign of the fraction (mantissa) (zero for positive and 1 for negative as before). The next seven bits contain a value which indicates the number of places that the hexadecimal point must be shifted to the right or left to obtain the correct order of magnitude. The method for determining the direction of the shift is explained in the next example. The remaining 24 bits contain the value of the mantissa. Exactly the same format is used with the doubleword field length, except that the additional 32 bits are allocated to the fractional portion, so that a total of 56 bits is available for that purpose. Since the use of 16 as a base is intrinsic in the design of the circuitry, it is not necessary to specify this base as part of the representation.

To gain an understanding of the actual representation of numbers in System/360 floating-point format, let us develop this expression for the value of 39_{10} (27_{16}) used above. We shall develop the representation for the fullword field length.

1. As a convenient starting point we shall again express the number in hexadecimal scientific notation:

$$.27_{16} \times 16^2$$

2. Next we shall construct the 24-bit fractional portion of the fullword. Except for the sign, which is taken care of in the very first bit, System/360 stores the mantissa portion of a floating-point number in packed-decimal format. That is, each half-byte accommodates one digit. This is still possible, of course, since there is sufficient room in a half-byte to handle any single hexadecimal digit. Thus, our fraction ($.270000_{16}$) is stored as

$$\overbrace{2}\ \overbrace{7}\ \overbrace{0}\ \overbrace{0}\ \overbrace{0}\ \overbrace{0}$$
$$0010\ 0111\ 0000\ 0000\ 0000\ 0000$$

3. Now the characteristic portion is constructed. The method used in System/360 allows further simplification by eliminating the necessity for an exponent sign. This is done by adding a large constant to the exponent value so that the resulting characteristic will always be positive. As with the base, the constant (System/360 uses a value of 64) is built into the circuitry and need not be explicitly given. The resulting characteristic (64 plus the desired exponent) is stored as a 7-bit binary number. In the case of our example this would be $(64 + 2) = 66_{10}$, or

   1000010

4. The remaining operation consists of specifying the proper sign (0 in this case) in the first bit and the resulting representation is

$$\overbrace{0}^{\text{sign}}\overbrace{1000010}^{66}\overbrace{00100111}^{2\ 7}\overbrace{0000000}^{0\ 0\ 0}\overbrace{0000}^{0}$$

A value of -39_{10} (or -27_{16}) would be represented exactly the same way with the exception of the sign bit which would be changed to 1. Additional examples are given in Table 2.9.

Doubleword floating-point representation would be the same as shown above, with the addition of another 32 bits at the right. Consequently, the use of a doubleword rather than a fullword provides the opportunity for greater precision in the fractional part but does not change the available range. Since 7 bits are allocated for the characteristic, its maximum value is 127 (i.e., $2^7 - 1$). Since the constant value of 64 is automatically added

TABLE 2.9 EXAMPLES OF NUMERICAL VALUES STORED IN FULLWORD FLOATING-POINT FORMAT

Decimal Value	Hexadecimal Floating-Point Equivalent			Internal Floating-Point Equivalent
31	$.1F \times 16^2$	0	1000010	00011111000000000000000
−629	$-.275 \times 16^3$	1	1000011	00100111001100000000000
72	$.48 \times 16^2$	0	1000010	01001000000000000000000
−11	$-.B \times 16^1$	1	1000001	10110000000000000000000
3212	$.C8C \times 16^3$	0	1000011	11001000110000000000000
511	$.1FF \times 16^3$	0	1000011	00011111111100000000000
.100	$.1999 \times 16^0$	0	1000000	00011001100110010000000
1000	$.3E8 \times 16^3$	0	1000011	00111110100000000000000
.06	$.0F5C \times 16^{-1}$	0	0111111	00001111010111000000000
−.022	$-.5A2 \times 16^{-1}$	1	0111111	01011010001000000000000

to the exponent, this gives us a maximum exponent value of 63, meaning that the mantissa portion of a floating-point number in System/360 may be multiplied by as much as 16^{63}. Similarly, subtracting 64 from the smallest possible characteristic value (0) indicates that a mantissa may be multiplied by a quantity as small as 16^{-64}.

PROBLEMS

1. List the field sizes handled by System/360 instructions.
2. Define byte, halfword, fullword, and doubleword.
3. Registers 1-15 of a 360 processor contain the base addresses tabulated below:

Register Number	*Base Address*
1	671
2	20422
3	12
4	249
5	400
6	3120
7	165208
8	96
9	420
10	9
11	60606
12	12000
13	44
14	670
15	760

   (a) If an instruction has a base of 8 and a displacement of 96 what is the effective address?
   (b) What is the effective address if an instruction has a base of 4 and a displacement of 6?
   (c) Which register is being referred to by the base if the displacement is 400 and the effective address is 820?
   (d) What is the effective address when the index is 12, the base is 12 and the displacement is 670?
   (e) What is the value in the index portion of the instruction when the base is 9, the displacement is 760 and the effective address is 1429?
   (f) What is the displacement when the effective address is 7200, the base is 14 and the index 10?
4. Convert the following decimal values to hexadecimal.
   (a) 41 (b) 12 (c) 307 (d) 200 (e) 1240
   (f) 144 (g) −23 (h) 511 (i) 49 (j) 3600
5. Convert the following hexadecimal values to decimal.
   (a) 14 (b) 26 (c) F3 (d) CB (e) 1C2
   (f) B13 (g) C5 (h) 70 (i) B0 (j) E01

6. Express the following binary values in hexadecimal form.
   (a) 110110010111 (b) 000110110110 (c) 101011110101
   (d) 011100111111 (e) 100001000011
   (f) 101100000001100011101010
   (g) 000000000010001100011111
   (h) 010101011010101010001111
   (i) 000000000010001111111111
   (j) 100000000000000000000001
7. Express the following hexadecimal values as 24-bit binary numbers.
   (a) 12ED03 (b) CA4279 (c) 647F (d) FF
   (e) E47E (f) D0DA24 (g) CACCAC (h) 748
   (i) 1B (j) 23CD
8. Show how the following integers would be stored in zoned-decimal format, using EBCDIC conventions.
   (a) 36 (b) +744 (c) 11 (d) −485 (e) 20000
   (f) −64 (g) −640 (h) +329 (i) +1122
   (j) −42
9. Give the decimal and hexadecimal forms for the following zoned-decimal values. (Use EBCDIC and assume all values are integers.)
   (a) 11011011 (b) 11111001 11110110 11001001
   (c) 11110000 (d) 11111101 11110001 11011101
   (e) 11111011 11110110 11110001 11111000
10. Show the following values in packed-decimal form.
    (a) 79 (b) +611 (c) −309 (d) 20001
    (e) −424 (f) +101 (g) +60724 (h) −9090
    (i) 927 (j) +4
11. Show the following hexadecimal integers in packed-decimal form.
    (a) B4 (b) −1A1 (c) +204 (d) C2C
    (e) +2DD2 (f) −36D (g) 679 (h) 9F4
    (i) 20001 (j) +3E3E
12. Bearing in mind that a packed-decimal field can be up to 16 bytes long, what are the largest and smallest values that may be expressed in this format? (Assume that the imaginary decimal point may be anywhere in the field.)
13. Express the following values as binary halfwords in two's complement notation.
    (a) -30_{10} (b) -30_{16} (c) −4 (d) −2B
    (e) -17_{16} (f) -41_{10} (g) −12C (h) -126_{10}
    (i) -111_{16} (j) -111_{10}
14. State the largest positive and negative integers that can be expressed in (a) halfword binary form, (b) fullword binary form.
15. Express the following decimal values in scientific notation, using base 16.
    (a) 300 (b) −12.6 (c) 0.625 (d) 241
    (e) −.0312 (f) 2147 (g) 18.75 (h) 3.5
    (i) 11 (j) 111
16. Express the values in Problem 15 as fullword floating-point values.

Chapter 3

Basic Instruction Handling

System/360's emphasis on versatility and universality of operations is as evident at the instruction level as it is when examined from a broader organizational viewpoint. We shall explore this characteristic further in this chapter by examining the basic types of activities performed by System/360 processors, the types of machine instructions that are provided to carry out these operations, the anatomy of these commands, and procedures underlying their execution. This will give us a clearer and more detailed insight into the processor's capabilities and will provide an introduction to the Assembler Language.

3.1 BASIC INSTRUCTION FORMATS

In concept, the machine-oriented instructions provided by System/360 processors are similar to those available in earlier systems. That is, a command represents an imperative statement which calls for a specified operation (signified by an appropriate operation code) to be performed on the contents of one or more designated locations in memory (as indicated by the appropriate number of operands). Each type of operation is specified by a unique combination of bits which can be recognized and properly interpreted by the logical circuitry. The total collection of such recognizable

operation codes constitutes the processor's machine language. All instructions, regardless of the source language in which they are originally prepared, must eventually be transformed into a sequence of such codes, accompanied by properly specified operands.

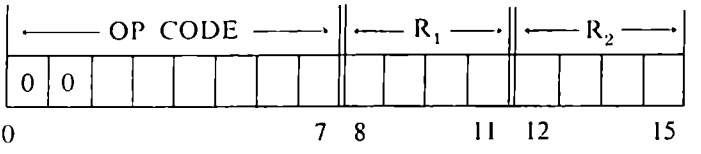
Once we get away from conceptual considerations we find that the 360 processor represents a considerable departure from its predecessors with respect to the organization of machine instructions. The basic difference lies in the fact that the processor must provide some practical means for working with information that is stored in the variety of forms described in the previous chapter. It turns out that the most convenient way to do this is to present the programmer with several sets of instructions, each of which constitutes a more or less complete repertoire for handling a particular type and number of operands. This is in direct contrast to most earlier processors, which were designed to provide a machine language whose commands were structured around a fixed number of operands. Thus, for example, the 360 machine language includes several instructions for a particular type of addition, with the use of a given instruction being dictated by the nature of the operands to be added.

Each of the operation codes in System/360 machine language is represented by a unique 8-bit binary string (i.e., one byte). For purposes of convenience we shall usually designate such instructions as equivalent 2-digit hexadecimal numbers. Thus the instruction 01011010, which represents a particular type of addition, can also be expressed as 5A.

There are five basic types of instruction formats which are implemented in three instruction sizes (1, 2, or 3 halfwords in length). Our first reaction to this variability in length and format might be to expect the processor to encounter added difficulties in decoding and analyzing such instructions; however, we shall see in the following sections that the machine language is organized in such a way as to facilitate the recognition of an instruction type and the subsequent determination of the number, nature, and location of the operand or operands to be processed.

3.1.1 The RR Format

The instructions in this set are each a halfword (16 bits) long and have the general form



The "RR" stands for "register-to-register" and indicates that this type of instruction involves the general-purpose registers, discussed briefly in the previous chapter. As shown in the schematic format diagram, the operation

code occupies the first eight bits (conventionally numbered 0–7), as it does in all instructions. Bits 8–11, designated R_1, and the final half-byte, designated R_2, contain the two operand addresses. Each of these addresses refers either to one of the 16 general-purpose registers or one of four *floating-point registers* which are also provided in System/360 processors. As stated previously, the general-purpose registers are numbered 0000_2 thru 1111_2 and each consists of 32 bits. In addition, there are four floating-point registers, each having a length of 64 bits. The identification numbers (or "addresses") assigned to these registers are 0000, 0010, 0100, and 0110. We shall learn more about their properties and uses later.

Since the floating-point register addresses have the same numerical values as those of some of the general-purpose registers, a mechanism must be provided to allow the processor's hardware/software complex to determine which register is being indicated. This is done through the operation code by having certain codes which take only floating-point registers as operands and others specifically intended for use with general-purpose registers. This technique is used for the more general purpose of providing means for distinguishing among instruction types. Specifically, the first two bits (0–1) of the operation code indicate the instruction type. For RR instructions, these two bits are always 00. Otherwise stated, hexadecimal representation of RR instructions will always begin with either 0, 1, 2, or 3.

As an example of RR construction let us look at

0001	1010	0010	0100
1	A	2	4

The 0001 in bits 0–3 specify that this is an RR instruction and also indicates that the operands are general-purpose registers. This particular instruction (1A) calls for addition of the contents of general register 4 to those of general register 2.

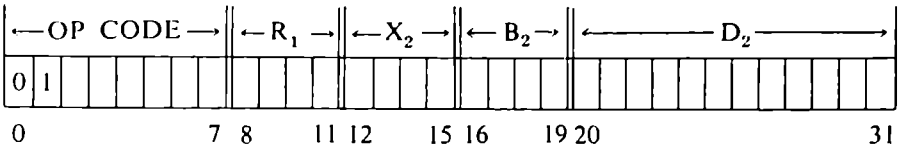
The instruction

0010	1010	0010	0100
2	A	2	4

also specifies an addition. The first hexadecimal digit (2) still labels it as an RR instruction but it also tells the processor to treat the operand values (2 and 4) as floating-point register addresses whose contents are floating-point values.

3.1.2 The RX Format

This set of instructions (known as register-and-indexed storage) allows the programmer to manipulate data between a register and main storage by specifying operands of each kind. The general form, shown below,



calls for the operation code in bits 0–7 (as always), a register address (either general-purpose or floating-point) in bits 8–11 and a storage address in the remaining portion of the instruction word. The three components of that address (base, index, and displacement) are used as described in the previous chapter to produce an effective address.

Identification of RX instructions is established by the first two bits of the operation code, said bits always being 01. Thus the hexadecimal representation of RX instructions always begins with 4, 5, 6, or 7.

To illustrate, let us consider the instruction

OP CODE		R_1	X_2	B_2	D_2		
0100	1010	0110	0011	0101	1010	1101	0111
4	A	6	3	5	A	D	7

The 4 in bits 0–3 labels the instruction as an RX type and specifies that R_1 refers to a general-purpose register. This happens to be an addition instruction. Its execution would result in calculation of the second operand address as D_2(AD7) plus the contents of the last 24 bits of general-purpose register 5 plus the last 24 bits of general-purpose register 3. The contents of the address specified by that sum would then be added to those in general-purpose register 6.

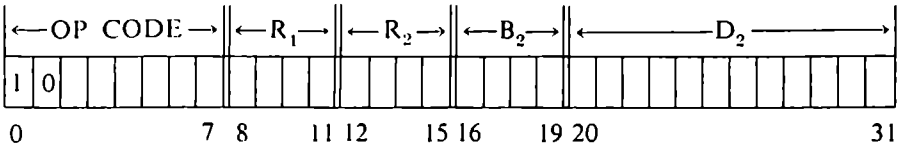
Changing the operation code to $6A_{16}$, so that we had

0110	1010	0110	0011	0101	1010	1101	0111
6	A						

would produce the same action, except that the contents of the calculated address would be treated as a floating-point number and they would be added to those of floating-point register 6.

3.1.3 The RS Format

This group, known as register-and-storage instructions, uses basically the same format as the RX type, with one basic exception as shown:



The index register designation (X_2) is replaced by R_3, which refers to a third register and does not participate in the calculation of an effective operand address. Instead, it is intended to provide the ability to work with several registers in one command. All RS operation codes begin with 10.

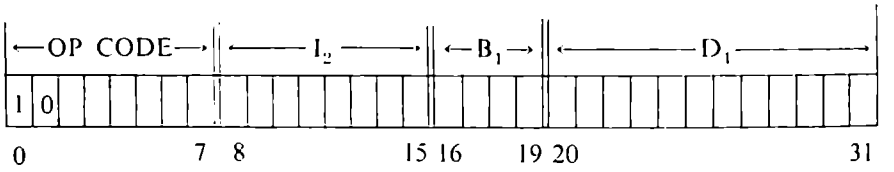
For example, the instruction

OP CODE		R_1	R_3	B_2	D_2		
1001	0000	0011	0111	0100	0110	1001	0111
9	0	3	7	4	6	9	7

is known as a Store Multiple command and, in essence, says the following: "Store a group of values from consecutive general-purpose registers starting with the one designated in R_1 and going through the one given in R_3, using consecutive memory locations, starting with the effective address calculated from B_2 and D_2." In our specific example, the contents of general-purpose registers 3 through 7 would be stored in the five consecutive fullword locations beginning with the one obtained by adding the contents of general-purpose register 4 to the value $(697)_{16}$.

3.1.4
The SI Format

SI (for storage-and-immediate-operand) instructions are two-operand instructions having the same length as the RX and RS types. The general form is shown below.



The first operand is addressed by base register B_1 and displacement D_1. The second operand consists of the *actual value* in bits 8–15. That is, no operand address is required, since the value is in the instruction itself. Operands of this type are called *immediate operands*. The interpretation of the byte contained in an SI instruction's immediate operand depends on the particular command involved. The first two bits of the operation code are the same as for the RS types.

An example of an SI instruction is

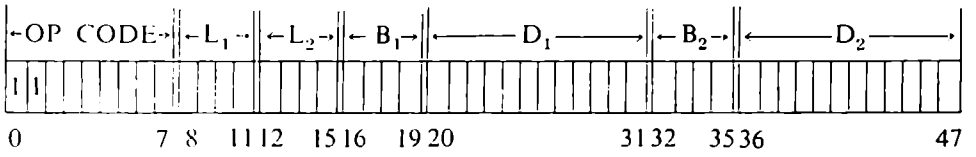
OP CODE		I_2		B_1	D_1		
1001	0010	1111	0000	1111	0000	0111	1000
9	2	F	0	F	0	7	8

This is a Move instruction, which causes the contents of the location specified by the first operand to be replaced by the second operand. For this specific

illustration, the byte at the address calculated by adding $(078)_{16}$ to the contents of general-purpose register F (register $(15)_{10}$) would be replaced by 11110000.

3.1.5 The SS Format

This last format (storage-to-storage) is used for manipulating two operands contained in main storage. Since each of these can be located by specifying the ingredients of an effective address, the length of an SS instruction is necessarily greater than that required for the other types. The general form is shown:



The address of the first operand is calculated from D_1 and the contents of the general-purpose register specified in B_1, as would be expected, with the second address being determined the same way. In addition, the SS format includes a unique feature (L_1 and L_2) which allows the programmer to specify the respective field lengths of the two operands (expressed in bytes). Furthermore, if both field lengths are to be the same, certain SS instructions allow the programmer to use the combined lengths of L_1 and L_2 as a single field to specify the common operand length.

Since the programmer using an SS instruction intends his operands to be at least one byte long, an L_1 and/or L_2 specification of 0000 is meaningless. Consequently, the four bits provided in L_1 or L_2 would ordinarily allow the specification of field lengths from 1-15 bytes. However, the inclusion of special circuits extends this by adding one to each of the binary numbers given by L_1 and L_2.

For example, the instruction

OP CODE		L_1	L_2	B_1	D_1			B_2	D_2		
1111	1010	0100	0011	1110	0110	1001	1101	0110	0000	1011	1010
F	A	4	3	E	6	9	D	6	O	B	A

is an SS Add instruction which will do the following in this specific case: The second operand, (3 + 1) bytes long and located beginning in the address given by adding $(0BA)_{16}$ to the contents of general-purpose register 6, is added to the (4 + 1) byte field beginning in the address calculated by the sum of $(690)_{16}$ and the contents of register 14.

As the reader may have inferred, the SS type of instruction is generally for decimal operations usually performed on packed-decimal operands. Their use of general-purpose registers is restricted to address calculation.

This is in contrast to RR and RX instructions which use these and the floating-point registers for accumulating and storing results.

3.2 BASIC INSTRUCTION TYPES

In the foregoing section we examined the five instruction formats and found that the machine language is organized along certain functional lines to facilitate processing and to provide the programmer with great flexibility. Specifically, we established the fact that registers and/or main memory, in virtually any combination, can serve as operands, depending on the needs of a particular situation. In addition, we saw that the format for a given instruction is signaled by the first two bits of the operation code. The processing circuitry actually examines these bits first and uses them to determine the number of bytes to access for that particular instruction. With respect to the machine language in general, its organization reflects the systematic assignment of operation codes along functional lines. This is seen in the summary given in Table 3.1.

The format represents only one way of categorizing System/360 machine instructions. Depending on the intended purpose, it is possible to define a large variety of arbitrary classification schemes, each of which may illustrate a different point. We shall be interested in two such categorizations: one that classifies the instructions according to the kind of operation being performed, and a second scheme that concerns itself with the nature of the operand on which the processing is being performed.

3.2.1 Classification by Operand

We shall dispense with the latter classification system at this stage merely by indicating that instructions are available for the following types of operands.

1. *Bytes.* It is possible to manipulate individual bytes, treating the contents of each one as a single character.
2. *Halfwords.* A variety of instructions is available in which the operands are expected to be halfwords. When such instructions are concerned with performing operations on numbers (such as addition or subtraction) they expect the operands to be in binary form.
3. *Fullwords.* The halfword operations are, in general, duplicated by another subset of the instruction repertoire intended for fullwords. Again, when the operand is numerical, it is expected to be in binary form.
4. *Packed Data.* The instruction categories mentioned thus far are combined in a set known as the *standard set.* (Table 3.1 summarizes the machine-language instructions and indicates whether or not each belongs to this set). Optional additions to the processor's hardware/

TABLE 3.1 SYSTEM/360 MACHINE-LANGUAGE INSTRUCTIONS—GENERAL SUMMARY

Description	Binary Operation Code	Hexadecimal Operation Code	Format	Functional Type	Set
Add	00011010	1A	RR	Fixed-point arithmetic	Standard
Add	01011010	5A	RX		
Add Halfword	01001010	4A	RX		
Add Logical	00011110	1E	RR		
Add Logical	01011110	5E	RX		
AND	00010100	14	RR	Logical operations	
AND	01010100	54	RX		
AND	10010100	94	SI		
AND	11010100	D4	SS		
Branch and Link	00000101	05	RR	Branching operations	
Branch and Link	01000101	45	RX		
Branch on Condition	00000111	07	RR		
Branch on Condition	01000111	47	RX		
Branch on Count	00000110	06	RR		
Branch on Count	01000110	46	RX		
Branch on Index High	10000110	86	RS		
Branch on Index Low or Equal	10000111	87	RS		
Compare	00011001	19	RR	Fixed-point arithmetic	
Compare	01011001	59	RX		
Compare Halfword	01001001	49	RX		
Compare Logical	00011001	15	RR	Logical operations	
Compare Logical	01010101	55	RX		
Compare Logical	11010101	05	SS		
Compare Logical	10010101	95	SI		
Convert to Binary	01001111	4F	RX	Fixed-point arithmetic	
Convert to Decimal	01001110	4E	RX		

TABLE 3.1 (Continued)

Description	Binary Operation Code	Hexadecimal Operation Code	Format	Functional Type	Set
Diagnose	10000011	83	SI	Status-switching	Standard
Divide	00011101	1D	RR	Fixed-point arithmetic	
Divide	01011101	5D	RX		
Exclusive OR	00010111	17	RR	Logical operations	
Exclusive OR	01010111	57	RX		
Exclusive OR	10010111	97	SI		
Exclusive OR	11010111	D7	SS		
Execute	01000100	44	RX	Branching operations	
Halt I/O	10011110	9E	SI	Input/output	
Insert Character	01000011	43	RX	Logical operations	
Load	00011000	18	RR	Fixed-point arithmetic	
Load	01011000	58	RX		
Load Address	01000001	41	RX	Logical operations	
Load and Test	00010010	12	RR	Fixed-point arithmetic	
Load Complement	00010011	13	RR		
Load Halfword	01001000	48	RX		
Load Multiple	10011000	98	RS		
Load Negative	00010001	11	RR		
Load Positive	00010000	10	RR		
Load PSW	10000010	82	SI	Status-switching	
Move	10010010	92	SI	Logical operations	
Move	11010010	D2	SS		
Move Numerics	11010001	D1	SS		
Move with Offset	11110001	F1	SS	Decimal arithmetic	
Move Zones	11010011	D3	SS	Logical operations	
Multiply	00011100	1C	RR	Fixed-point arithmetic	
Multiply	01011100	5C	RX		
Multiply Halfword	01001100	4C	RX		

TABLE 3.1 (Continued)

Description	Binary Operation Code	Hexadecimal Operation Code	Format	Functional Type	Set
OR	00010110	16	RR	Logical operations	Standard
OR	01010110	56	RX	Logical operations	
OR	10010110	96	SI		
OR	11010110	D6	SS		
Pack	11110010	F2	SS	Decimal arithmetic	
Set Program Mask	00000100	04	RR	Status-switching	
Set System Mask	10000000	80	SI		
Shift Left Double	10001111	8F	RS	Fixed-point arithmetic	
Shift Left Single	10001011	8B	RS		
Shift Left Double Logical	10001101	8D	RS	Logical operations	
Shift Left Single Logical	10001001	89	RS		
Shift Right Double	10001111	8E	RS	Fixed-point arithmetic	
Shift Right Single	10001010	8A	RS		
Shift Right Double Logical	10001100	8C	RS	Logical operations	
Shift Right Single Logical	10001000	88	RS		
Start I/O	10011100	9C	SI	Input/output	
Store	01011101	50	RX	Fixed-point arithmetic	
Store Character	01000010	42	RX	Logical operations	
Store Halfword	01000000	40	RX	Fixed-point arithmetic	
Store Multiple	10010000	90	RS		

TABLE 3.1 (Continued)

Description	Binary Operation Code	Hexadecimal Operation Code	Format	Functional Type	Set
Subtract	00011011	1B	RR	Fixed-point arithmetic	Standard
Subtract	01011011	5B	RX		
Subtract Halfword	01001011	4B	RX		
Subtract Logical	00011111	1F	RR		
Subtract Logical	01011111	5F	RX		
Superviser Call	00001010	0A	RR	Status-switching	
Test and Set	10010011	93	SI	Logical operations	
Test Channel	10011111	9F	SI	Input/output	
Test I/O	10011101	9D	SI		
Test Under Mask	10010001	91	SI	Logical operations	
Translate	11011100	DC	SS		
Translate and Test	11011101	DD	SS		
Unpack	11110011	F3	SS	Decimal arithmetic	
Read Direct	10000101	85	SI	Status-switching	Direct control feature
Write Direct	10000100	94	SI		
Insert Storage Key	00001001	09	RR		Protection feature
Set Storage Key	00001000	08	RR		
Add Decimal	11111010	FA	SS	Decimal arithmetic	Decimal feature
Compare Decimal	11111001	F9	SS		
Divide Decimal	11111101	FD	SS		
Edit	11011110	DE	SS	Logical operations	
Edit and Mark	11011111	DF	SS		

TABLE 3.1 (Continued)

Description	Binary Operation Code	Hexadecimal Operation Code	Format	Functional Type	Set
Multiply Decimal	11111100	FC	SS	Decimal arithmetic	Decimal feature
Subtract Decimal	11111011	FB	SS		
Zero and Add	11111000	F8	SS		
Add Normalized (Long)	00101010	2A	RR	Floating-point arithmetic	Floating-point feature
Add Normalized (Long)	01101010	6A	RX		
Add Normalized (Short)	00111010	3A	RR		
Add Normalized (Short)	01111010	7A	RX		
Add Unnormalized (Long)	00101110	2E	RR		
Add Unnormalized (Long)	01101110	6E	RX		
Add Unnormalized (Short)	00111110	3E	RR		
Add Unnormalized (Short)	01111110	7E	RX		
Compare (Long)	00101001	29	RR		
Compare (Long)	01101001	69	RX		
Compare (Short)	00111001	39	RR		
Compare (Short)	01111001	79	RX		
Divide (Long)	00101101	2D	RR		
Divide (Long)	01101101	6D	RX		
Divide (Short)	00111101	3D	RR		
Divide (Short)	01111101	7D	RX		
Halve (Long)	00100100	24	RR		
Halve (Short)	00110100	34	RR		
Load and Test (Long)	00100010	22	RR		
Load and Test (Short)	00110010	32	RR		
Load Complement (Long)	00100011	23	RR		

TABLE 3.1 (Continued)

Description	Binary Operation Code	Hexadecimal Operation Code	Format	Functional Type	Set
Load Complement (Short)	00110011	33	RR	Floating-point arithmetic	Floating-point feature
Load (Long)	00101000	28	RR		
Load (Long)	01101000	68	RX		
Load Negative (Long)	00100001	21	RR		
Load Negative (Short)	00110001	31	RR		
Load Positive (Long)	00100000	20	RR		
Load Positive (Short)	00110000	30	RR		
Load (Short)	00111000	38	RR		
Load (Short)	01111000	78	RX		
Multiply (Long)	00101100	2C	RR		
Multiply (Long)	01101100	6C	RX		
Multiply (Short)	00111100	3C	RR		
Multiply (Short)	01111100	7C	RX		
Store (Long)	01100000	60	RX		
Store (Short)	01110000	70	RX		
Subtract Normalized (Long)	00101011	2B	RR		
Subtract Normalized (Long)	01101011	6B	RX		
Subtract Normalized (Short)	00111011	3B	RR		
Subtract Normalized (Short)	01111011	7B	RX		
Subtract Unnormalized (Long)	00101111	2F	RR		
Subtract Unnormalized (Long)	01101111	6F	RX		
Subtract Unnormalized (Short)	00111111	3F	RR		
Subtract Unnormalized (Short)	01111111	7F	RX		

software allow the inclusion of another group (called the *decimal set*) which operates on packed data. These are generally in the SS format, thus permitting the specification of variable length fields.

5. *Floating-Point Operands.* A second optional group, called the floating-point set, allows the programmer to represent and process numerical data in floating point form. Duplicate instructions are provided to handle short (fullword) and long (doubleword) operands.

3.2.2 Functional Classification

For our present purposes we shall find it convenient to treat the repertoire of machine-language instructions as consisting of six basic types, based on the general kind of operation being performed. We shall not be concerned here with a description of each individual instruction. This allows us to avoid the difficulty caused by the fact that the assignment of certain instructions to certain specific categories in preference to others is debatable in this or any other arbitrarily defined classification scheme.

1. Shifting and Internal Data Transfer. Instructions in this category handle the transfer of data between various areas in storage. The repertoire is quite broad and includes commands for transfers between storage locations, registers, from storage to register, and vice versa. Operands may be of several lengths and in several formats, since separate instructions are included in all five formats. It is also possible to transfer an address, rather than its contents, to a desired internal destination.

   In addition, the programmer can call for a transfer of information within a given location. Instructions are provided as part of the standard machine language for shifting individual bits in a byte.
2. Arithmetic Operations. The basic operations of addition, subtraction, multiplication, and division are provided for all allowable types of operands and data formats. In addition, instructions are available for generating the complement of a number and halving a number. (Although the latter represents division by a specific divisor, it can be done much faster by treating it separately, since halving of a binary number is achieved by shifting the bits one position to the right, assuming a fixed binary point.)
3. Logical Operations and Decisions. This group includes an extensive set of instructions for performing Boolean operations (such as AND and OR) on binary strings. In addition, there is a large number of commands for performing various types of internal tests in preparation for decisions based on their outcome. Such tests include the equality of two data fields, the presence or absence of certain signals, and the status of certain indicators.
4. Data-Conversion Instructions. In order to allow the programmer to exploit the flexibility of the various instruction formats and types, a

group of instructions is provided to allow convenient mediation among data forms. Thus the programmer is able to go between decimal and binary, packed and unpacked, and fixed and floating point by means of simple specifications. In addition, he can call upon very powerful instructions to make extensive symbol substitutions, replacing entire arrays of numbers or characters with others in one programming step.

5. Input/Output. Since all data transmission activities are handled by the I/O channels, the machine language instructions dealing with this aspect of processing are small in number and very general. Basically, these instructions serve merely to initiate, cut off or monitor I/O operations, with the channel programs handling the actual transmission. Consequently, the instructions include extensive provisions for specifying the particular device involved.
6. Sequence and Status Changing. In many cases the decision engendered by the outcome of a test (such as performed by an instruction in group 3 above) involves a change in processing sequence, i.e., transfer to a group of instructions in some other portion of the program and a resumption of execution at that point. A number of different types of sequence changes is possible, depending on the decision criteria used and on a number of other factors. Several instructions (commonly called *branching instructions*) are included to handle these contingencies.

Because of its complexity and versatility, System/360 design includes a number of built-in indicators which reflect the current status of various aspects of processing. Correspondingly, instructions are provided to test these indicators and, where appropriate, to place the processor in a particular status. For example, a processor waiting for input is in a conceptually different status from one currently receiving input, and a certain indicator would reflect this. Since the hardware and supervisory software check and act on many of the same indicators examined by the programmer, a change in status may often be effected by using an instruction which adjusts an appropriate indicator.

As stated before, a number of alternatives exist (and more can be devised) for classifying the System/360 machine language. Additional categorizations are given in Table 3.1 and in the manufacturer's literature[1].

3.3 PROCESSING OF INSTRUCTION SEQUENCES

Because of System/360's basic versatility, the sequence of events which comprises the execution of even a simple program is substantially more complex than has been the case in most previous processing systems. Perhaps the most distinguishing characteristic of this processing is the constant interaction among the hardware, software and the user's program (which we shall call the *problem program*), resulting in a large number of

supporting activities that are beyond the user's direct cognizance and control. Many of these represent automated versions of activities specified by the programmer, when dealing with earlier systems, while others are necessitated by the system's operating characteristics.

In order to acquire a good understanding of the conceptual mechanisms inherent in System/360 instruction processing, it will be necessary for us to examine the operating environment provided for various types of programs. Toward this end we shall concern ourselves here with the basic method for storing instructions and the use of certain machine indicators as continuous monitors of the system's activity.

3.3.1 Normal Sequential Instruction Processing

System/360 problem programs may be executed from anywhere in main storage. As pointed out previously, this flexibility is made possible by the assignment of instruction addresses relevant to some reference point, rather than as absolute locations, with the specification of that reference point being made just prior to execution time. At that time each instruction address is adjusted by the appropriate amount (by adding the reference value to it) and the program is loaded into storage. We shall concern ourselves with the conduct of events subsequent to that point.

Let us consider the following schematic illustration of a portion of a program stored in memory and about to be executed.

Instruction Location (Hex.)	*Instruction Format Type*	*Instruction*
12000	RX	01111011001000000001000000100111
12004	RX	01111010001000000001000001101001
12008	RR	0011000000110100
1200A	SS	10110001000100010100000000010011000010000001010111
12010	SI	10010101110111010100000000010011
		.
		.
Final instruction of problem program		

Let us assume that the instruction located at $(12000)_{16}$ represents the first instruction of the program. Furthermore, we shall not concern ourselves with how the program gets started, beyond the statement that the supervisory software, having seen to it that the program is properly loaded, turns control over to it by directing the attention of the appropriate circuits to the first instruction.

The first instruction is in the RX format and begins in location 12000. Since it is four bytes long, it occupies locations 12000–12003, with the second instruction beginning at 12004. In order for the processing circuitry to allow the proper execution of the program, it must keep track of its activities. In most processors this is done by setting an instruction counter to the address of the first instruction and continually changing the value in the counter so that its contents always reflect the particular instruction being worked on (or about to be worked on) at a given point in time. System/360 uses such an instruction counter, along with a number of other activity indicators. For the present we shall use this instruction counter conceptually, without reference to its physical counterpart in the system.

With the instruction counter initially set at 12000, the control circuitry scans the operation code and, by examining its first two bits, determines the length of the instruction. The appropriate number of bytes is accessed (12000–12003 in this case) and the instruction length is added to the instruction address, thus updating the counter. The instruction is then interpreted and the necessary circuits are activated toward its execution. The value in the instruction counter is then used to access the next instruction, determine its length, update the counter, and so on. This continues until execution of the problem program is completed (at which point control reverts to the supervisor) or until this normal sequence is broken.

3.3.2 Branching

The programmer has the ability to alter the normal sequence of execution by specifying that the next instruction to be executed at a given point in his program is to be fetched from a location not adjacent to that of the last instruction processed. This is accomplished by one of a variety of branch instructions, which indicates the next location to be accessed and forces sequential processing to proceed from that point. Operationally, we can think of such a branch as an instruction which, when interpreted by the control circuitry, causes the instruction address calculation process to be suspended, with a branch address being placed in the instruction counter instead. Once such a branch has been executed, processing continues in the normal sequential manner.

3.3.3 Interrupts

There are countless situations which will precipitate a break in the normal processing sequence. In some cases the type of break and its point of occurrence in the program can be predicted, but its time of occurrence relative to the start of processing cannot, while many other cases are completely unanticipated. In either event, these are sequence changes which cannot be handled (except in the most inefficient way) by branch instruc-

tions. Such situations are called interruptions (*interrupts* in the jargon) and must be handled automatically (outside of the domain of the problem program) if the system's performance features are to be exploited to any appreciable extent. System/360 provides a very elaborate hardware/software complex for detecting interruptions in problem programs, switching rapidly to special programs designed to take appropriate action and, upon completion of their execution, switching back to the problem program at the point of interruption.

The occurrence of an interruption does not necessarily mean that something is wrong. Certain types of interruptions are routine events in any processing sequence. For example, the execution of an I/O operation requires several interruptions because of the logical relationship between the central processor and the I/O channels. Remember that the problem program cannot give specific I/O instructions—it can only indicate that a certain type of I/O operation is desired. Once the control circuitry recognizes this, an interrupt must occur in order for the supervisor to take control long enough to initiate the execution of the channel program that will actually handle the data transmission. Control can then be returned to the problem program, which resumes processing while the I/O operation continues independently. When this operation is complete, the channel automatically informs the central processor of this, thus allowing the supervisor to initiate another I/O operation (if one is waiting to be done), thereby making maximum use of the overlap capabilities.

3.3.3.1 Classes of Interrupts

The many types of interrupts recognized and handled by the system may be conveniently grouped into five basic classes. We shall merely make mention of them here, deferring any detailed explanations to subsequent sections in which we will actually be working with them.

1. *I/O Interrupts.* These are interrupts caused by conditions relating to data transmission operations. For instance, the fact that the execution of a channel program has been completed and the channel is free will generate an I/O interrupt, with subsequent system activity being dictated by the conditions in the channel or the I/O unit.
2. *Program Interrupts.* These are caused by any one of a variety of "unusual" conditions which can arise during the execution of a problem program. Such an interrupt will occur, for example, if the control circuitry comes across an illegal instruction code (one not recognized as part of the machine's repertoire), or a reference to a nonexistent address.
3. *Supervisor Call Interrupts.* When certain conditions arise, it is possible for the problem program to "call for" an interrupt. This is achieved

by means of a special instruction to the supervisor, said instruction (a Supervisor Call) defining the type of interrupt and the consequent action to be taken.

4. *External Interrupts.* This type of interrupt can be caused manually by activating the interrupt key on the control panel. In addition, it can be precipitated by a signal from the timer, another connected central processing unit, or some other special device outside of the system's basic configuration.
5. *Machine Check Interrupts.* The mechanism which handles this type of interrupt is designed strictly for the recognition of a variety of hardware malfunctions.

These classifications are not arbitrary. Separate and distinct mechanisms, both in terms of special hardware circuits and software routines exist for recognizing and handling each class as an entity. Furthermore, instructions are provided for the programmer to address himself to the various types and, if he wishes, to provide his own routines for handling any or all interrupt types.

3.3.3.2 Masking of Interrupts

Although the mechanisms for recognizing and handling interrupts are intrinsic parts of the hardware/software complex, it is possible for the programmer to issue instructions which will deactivate certain interrupts. This process, known as *masking*, will suspend normal handling of selected interrupts, causing the system to behave temporarily as if those interrupts did not exist. Included among the interrupts that can be masked are those in the machine check class and several types of interrupts in the program class.

3.3.4 Machine Status

A third type of sequence change, known as *status-switching*, is very closely associated with the interrupt-handling process. System/360 is organized to operate under sets of alternate machine states, each of which is associated with certain hardware-related properties. There are four basic types of machine states, each of which has two alternatives:

1. *Problem/Supervisor.* The processor operates in either of these alternative states, with the superficial distinction that one of the supervisory programs, rather than a problem program, is being executed in the supervisor state. More specifically, there are certain types of machine instructions that cannot be recognized and processed while the machine is in the problem state. For example, an input/output instruction is a *privileged instruction.* That is, when an input/output instruction is encountered by the circuitry while the processor is in the problem state,

it is treated as an error and a program interruption will automatically occur, switching the machine status to the supervisor state.

2. *Wait/Running.* When the processor is in the *running* state, it is routinely fetching, analyzing, and executing instructions in normal fashion. The instructions being processed in this state may be part of a problem program or a supervisory one. This is in contrast to the *wait* state, in which case no instructions are being processed. The occurrence of certain types of interruptions can cause a corresponding change from one of these states to the other. For example, a Supervisor Call instruction (operation code 0A) encountered in a problem program will change the machine state from running to wait. Similarly, a processor currently in the wait state may have its status switched to the running state by an I/O or external interrupt.
3. *Masked/Interruptable.* It was indicated before that the programmer has the capability of temporarily deactivating (masking) certain interrupts. When he does this, the processor is placed in the *masked* state. Since it is impossible to mask all interruptions, the processor is always *interruptable* with respect to certain classes of interrupts. Switching between these two states is achieved by the use of appropriate general or privileged instructions.
4. *Stopped/Operating State.* These alternative states are somewhat similar (but by no means identical) to the running/waiting alternatives. The basic distinction lies in the fact that switching between them cannot be effected by an instruction or an interrupt. Only manual intervention or a machine malfunction can bring about stopped/operating status change. When the machine is in the stopped state, no instructions or interruptions can be executed, an appropriate light on the control panel indicates this and the timer stops operating. The machine reaches the stopped state for a very short period of time immediately after the last instruction of one program is executed and before the first instruction of the next one is brought in. It also goes into that state, of course, when somebody pushes the "stop" button on the control panel. Under the latter circumstances, depression of the "start" button will change the state back.

Each pair of alternatives is mutually exclusive, but this is not the case for the four types of states. That is, a complete description of the processor's status consists of a listing of four types of states. Thus, for example, a processor may be in the supervisor, masked, running, and operating states.

3.4 MACHINE INDICATORS

Since the recognition and processing of interrupts is a system function which is performed automatically, without the knowledge or direct intervention of the programmer, he can no longer be expected to supply

the type of information required for the satisfactory handling of such situations. It is up to the system, therefore, to determine the type of interruption that has occurred, what to do about it, and where to return control once the appropriate action has been completed.

This type of information, which is crucial to the operation of any processing system, changes very rapidly and must be accessible to a large number of hardware and software components for examination and modification. In System/360 processors, a great deal of such information is provided in a special set of machine indicators consisting of several reserved locations in main memory and a special memory section which has no address capabilities (as far as the programmer is concerned).

3.4.1 The Program Status Word

Basic control of all instruction sequencing is handled by means of a 64-bit (doubleword) indicator known as the *Program Status Word* (PSW). Information provided and maintained through the PSW includes masking specifications, identification of interruption sources, instruction address and other fields which indicate the status of the system in relation to the program currently being executed.

There is a special 64-bit area of storage called the *current PSW* which serves as the active control center. It is not an organizational part of the main memory, since it has no associated numerical address and cannot be specified directly as an operand in most instructions. Instead, the circuitry is designed to use it automatically as a clearinghouse for all status information, which may be transmitted from the hardware or from a supervisory or problem program.

3.4.1.1 General PSW Format

The current PSW is set up to be in constant communication with a number of permanently assigned groups of memory locations. Each of these auxiliary fields is given a particular PSW name and is reserved (by circuit design, not by convention) for a certain type of PSW information. When that information is required, the current PSW automatically seeks it from the appropriate PSW in main memory.

Let us make sure this is clear. There are several areas in main memory that are used to contain PSW information. Each of these reports to the current PSW and is organized exactly like it (same length and format). Their function is to supply status and control information to the current PSW (as either a partial or total change of its contents) with the control action emanating from the current PSW. We shall discuss the types of *auxiliary* PSW's in the next section, once we establish their organization.

The PSW is organized as shown in Table 3.2. We can outline the function of the various fields as follows:

TABLE 3.2 PROGRAM STATUS WORD ORGANIZATION

Bit Position	Contents
0	Multiplexer Channel (Channel 0) Mask
1	Selector Channel 1 Mask
2	Selector Channel 2 Mask
3	Selector Channel 3 Mask
4	Selector Channel 4 Mask
5	Selector Channel 5 Mask
6	Selector Channel 6 Mask
7	External Mask
8-11	Protection Key
12	ASCII Mode
13	Machine Check Mask
14	Wait State Indicator
15	Problem State Indicator
16-31	Interruption Code
32 33	Instruction Length Code
34-35	Condition Code
36	Program Mask
37	Decimal Overflow Mask
38	Exponent Underflow Mask
39	Significance Mask
40-63	Instruction Address

1. System Mask Bits (0-7). Each of the first eight bits is associated with an I/O channel or a set of external signals and can be used to control the processor's interruptability with respect to these sources. When a mask bit is 1, the corresponding source can interrupt the central processor. If it is 0, it cannot. The three types of external sources masked by bit 7 are the timer, the interrupt key and special external devices.
2. Protection Key (8-11). When a central processor is equipped with the memory protection feature, the accessibility of a designated portion of memory is controlled by this protection key. Under the storage protection arrangement, each 2,048-byte block of memory is serviced by a special five-bit register called a *storage key*. The contents of these keys are accessible to and modifiable by the supervisory software. When the control circuitry encounters an instruction to store something in memory, a comparison is automatically made between the first four bits of the associated storage key and the protection key. The storage operation will be executed only when the two fields match or when the protection key is zero. A mismatch will cause an interrupt.
3. ASCII mode Indicator (bit 12). When this bit is 1, the central processor will generate results and interpret data according to the ASCII Code (see Chapter 2). Otherwise, EBCDIC is used.
4. Machine Check Mask (bit 13). A value of 1 in this bit will suppress the signals and diagnostics that usually occur when there is a machine

malfunction. (This is of greater concern to the maintenance personnel than to programmers.)

5. Wait State Indicator (bit 14). When the wait state indicator contains a 1, the processor is in the wait state; 0 indicates the running state.
6. Problem State Indicator (bit 15). The supervisor state is indicated by a 0 in the bit (with 1 corresponding to the problem state).
7. Interruption Code (bits 16-31). For four of the five interruption classes (I/O, program, supervisor call, and external classes) there is a specified group of recognizable interrupts, each with a particular identifying code. This code, contained in the indicated PSW bits, allows the supervisory system to determine the source of the interruption. Twenty-four such codes are provided. The interruption code for a machine check interrupt is all zeros.
8. Instruction Length Code (bits 32-33). This portion of the PSW is used when a program or supervisor call type of interruption occurs. At that point, the length of the last-interpreted instruction (expressed as the number of halfwords) is given here. The contents are meaningless for the other three interrupt classes. Table 3.3 summarizes the instruction length codes:

TABLE 3.3 PSW INSTRUCTION LENGTH CODES

Format of Last-Interpreted Instruction	Instruction Length, Halfwords	Instruction Bits 0-1	PSW Bits 32-33
RR	1	00	01
RX	2	01	10
RS or SI	2	10	10
SS	3	11	11

9. Condition Code (bits 34-35). One of the most powerful control features in System/360 processors is embodied in the fact that the actual result produced by executing a machine instruction is usually accompanied by an indicator signal which shows up as 00, 01, 10, or 11 in bits 34-35 of the PSW. This allows the system to determine the type of general result. For example, when two numbers are added a condition code of 00 will be generated if the result is zero, 01 if the result is less than zero and 10 if it exceeds zero. A condition code of 3 (11_2) implies an overflow (a result too large to be completely accommodated in the specified or implied storage location).

   Similarly, a comparison of two numbers will set the condition code. 00 indicates equality, 01 says that the first operand is smaller and 10 represents a larger first operand. It is possible for the program to set up branches on condition codes.
10. Program Mask (36-39). It is possible for the programmer to mask four individual types of interruption in the program class. The masked

status for each of these is indicated by a 0 in the appropriate bit according to the following sequence:

Bit Number	*Type of Interrupt*
36	Overflow in fixed-point addition or subtraction.
37	Overflow in decimal addition or subtraction.
38	Exponent underflow (exponent of less than zero in a floating-point value).
39	Significance interrupt (a fraction of zero after floating-point addition or subtraction).

11. Instruction Address (bits 40–63). This field changes as each instruction is processed, so that it always contains the address of the first (or leftmost) byte of the next instruction.

3.4.1.2 Auxiliary PSW's

Now that we have examined the contents of the current PSW we can see how crucial its role is in the overall instruction-handling process. More specifically, we can now speak of the servicing of an interruption as a process in which the type and characteristics of the interruption are determined, appropriate instructions are executed and control is returned to the interrupted program at precisely the point where and under precisely the conditions prevailing when the interruption took place.

The information required for these manipulations is obtained by replacing the current contents of the PSW with another doubleword supplied to that indicator from one of several locations in main memory reserved for that purpose. These locations provide storage for "old" PSW's (those associated with interrupted programs) and serve as sources for "new" PSW information.

There are eleven doublewords reserved for PSW information (see Table 3.4). One of these is associated with initial program loading, five with "old" PSW's for each of the five interrupt classes and five with "new" PSW information.

We shall let the initial program loading PSW go for awhile and examine the conceptual sequence of events during a "typical" interruption. Let us suppose that sequential instruction processing is proceeding normally. While this is true, the current PSW is monitoring the operations, so that if an instruction is executed, bits 32–33 indicate its length and bits 40–63 contain the address of the next instruction. Now the sequence comes to a fixed point addition instruction and, as a result of its execution, there is an overflow.

TABLE 3.4 PSW STORAGE ASSIGNMENTS

Locations (Decimal)	Assignment	
0-7	Initial Program Loading PSW	
24-31	External	old PSW
32-39	Supervisor Call	
40-47	Program	
48-55	Machine Check	
56-63	I/O	
88-95	External	new PSW
96-103	Supervisor Call	
104-111	Program	
112-119	Machine Check	
120-127	I/O	

This circumstance (called an *exception*) causes an interrupt of the program sequence. Bit 36 of the current PSW is consulted to determine whether the fixed point interrupt is masked (let us assume it is not). Since the interruption requires a sequence change, the instruction located by bits 40–63 of the current PSW no longer applies (that is, until the interrupt is serviced). Consequently, the current PSW becomes the old PSW and, since we are dealing with a program interrupt, the contents are automatically sent to locations 40–47. By the same token, a new PSW is needed to direct the circuitry to the interrupt routine. This information is provided by loading the current PSW with the contents of the new program PSW (locations 104–111). Once the interruption routine is completed, the current PSW is loaded once again with the contents of the old program PSW and processing resumes.

The initial program loading PSW is used as a destination for control information when the system is first started (as it is at the beginning of an operating day). When the operator manually calls for initial I/O transmission, the information read in this manner is used for starting automatic processing.

3.4.2 Input/Output Indicators

The handling of input/output processes constitutes an aspect of System/360 instruction execution with which most programmers have little contact. Because of the wide range of possible I/O connections and the role of channels in the overall system organization, the handling of an I/O transaction is an inherently complex one. Before some information can actually be transmitted, the pertinent channel, device, and memory locations must be specified. In addition, the availability of the device and channel, as well as that of the processor, must be established.

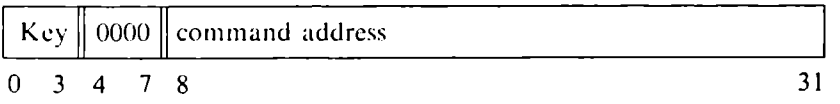
To simplify matters (as well as to allow overlapped operation), the task of handling I/O operations has been transferred from the programmer to the system. Once the programmer has indicated that he wishes to transmit (which he does by a "Start I/O" instruction) and has named the appropriate channel and device (also part of the instruction) the system takes over and does the rest. In fact, Start I/O will not be executed as part of the program. Instead, it causes an interruption which places the processor in the supervisor state, during which time the transaction is completed by executing the proper channel program.

We shall not be concerned with the detailed sequence of events which takes place during I/O operations. Our discussion will restrict itself rather to an overview of the interaction between the processor and a channel, so that the reader will acquire an understanding of the types of processes triggered by his instructions. Toward this end we shall examine several types of indicators which maintain and supply the information required for smooth and efficient I/O handling.

3.4.2.1 The Channel Address Word

When the circuitry encounters a Start I/O instruction, control is taken by the supervisor, which analyzes the Start I/O and turns it over to the channel specified in the instruction. (It is assumed, of course, that the channel and device have been checked by the system and found to be available.) The channel logic then consults a special fullword called the *Channel Address Word* (CAW) to determine what to do next.

The CAW is permanently assigned to locations 72–75 and has the following format:

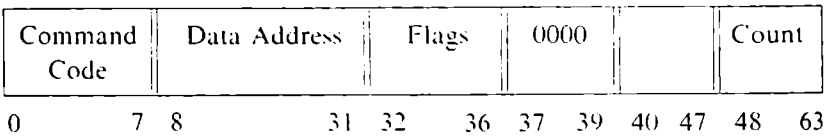


Bits 0–3 contain the protection key (see Sec. 3.4.1) for the storage location associated with this particular I/O operation; 4–7 must contain zeros and 8–31 contain an address in main memory whose contents constitute the first channel command. (We had better set down the nomenclature before we go any further. The program issues *instructions* which are decoded and handled by the processor. When control is given to a channel, it follows a sequence of one or more *commands*. The process of decoding these commands produces *orders* directed to the particular I/O device.)

Once the initial channel command has been located, the processor can be "reactivated" since the channel has enough information to continue processing independently. The CAW, whose contents have been reproduced in the channel, can now be altered for some other purpose.

3.4.2.2 The Channel Command Word (CCW)

Each command of a channel program is organized as a *Channel Command Word* (CCW) somewhere in the main memory. There are no restrictions (outside of the permanently assigned locations) with regards to the allowable number of locations of CCWs. When a Start I/O instruction is encountered, the CAW indicates the location of the first CCW, which contains an I/O command (not instruction) and an associated storage location. CCW format is as shown.



1. Bits 0-7 specify the operation to be performed. This is one of six commands which, strictly speaking, are not part of the machine language repertoire. These commands initiate an operation on a particular I/O device.
2. The data address (bits 8-31) specifies the location of the first byte of the area associated with the I/O operation. Note that the address is complete (not of the base-displacement type).
3. Each CCW contains five flag bits (bits 32-36) which can be set to cause an interrupt, terminate the I/O operation and control the use of subsequent CCW's.
4. Bits 37-39 must contain zeros (except under special conditions); 40-47 are not used.
5. The last 16 bits contain a value which represents the number of contiguous bytes of storage associated with this command. Depending on the contents of the flags, execution of this channel command will complete the processing for the problem program's I/O instruction, or will cause processing to continue by accessing the next CCW.

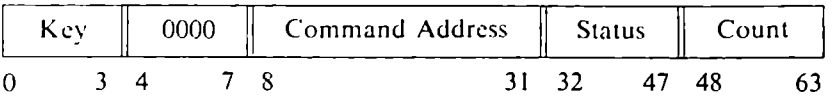
   In addition to these CCW's, which may be anywhere in main storage, there are two special CCW's for which locations 8-15 and 16-23 are permanently reserved. These CCW's are associated with the initial program loading process.

3.4.2.3 The Channel Status Word (CSW)

Locations 64-71 in main storage are reserved for the *Channel Status Word* (CSW) which serves as the vehicle for supplying status information about a device and channel after an I/O process has been completed. As part of channel program execution, the channel circuitry maintains

up-to-date status information in the CSW. Consequently, when the I/O process is completed (or otherwise terminated) and the resulting interruption occurs, the programmer can "consult" the CSW to determine the I/O status and the conditions under which termination occurred.

Although the CSW is called a status word, its format is somewhat different from that of a PSW:



1. The protection key (bits 0–3) is the same as that first supplied to the channel by the CAW (see Sec. 3.4.2.1) as the result of a Start I/O instruction.
2. Bits 4–7 must be zeros.
3. Bits 8–31 contain the address of the last CCW used by the channel.
4. Bits 32–47 are handled as two bytes, the first of which gives the I/O device's status and the second supplies channel status information.
5. The final 16 bits contain a value equal to the difference between the number of bytes specified in the count field of the last CCW and the actual number of bytes transmitted during the I/O operation.

Information in the CSW is available until the next I/O instruction is encountered, or any other I/O interrupt occurs, at which time part or all of the CSW can change.

PROBLEMS

1. Identify the format of the following instructions:
   (a) 01111010010010111001011010110111
   (b) 01000100010111001000000010110100
   (c) 11010101001100100011100010011001001000010100110
   (d) 0010101010100001
   (e) 0000011110111011
   (f) 10000110011010010001000000110101
   (g) 10010101100010110001000100110100
   (h) 01011011000010110011000101010111
   (i) 10000010100001000110000110110101
   (j) 10011000000110000101100010100001
2. For each of the ten machine-language instructions that follow, compute the effective addresses of each operand as a decimal number, assuming the contents of General Registers 5, 6, and 7 are as follows:

   GR5 00110000000000000000101111010001
   GR6 01001111000000000000110000111101
   GR7 10010110000000000000011100001001

(a) 0010101000100110
(b) 10010000010011110101000010100000
(c) 01001110010001110110000000000000
(d) 01010000000100000101000011110000
(e) 10010010111100000111000000001001
(f) 10010100010010010000010000000000
(g) 10001001001010100110100000000000
(h) 01111010001101100110000000001000
(i) 10010111000000000101000010100001
(j) 10001001101010100111010110010111

3. What is the conceptual difference between a branch and an interruption?
4. Describe the five classes of interruptions.
5. What is meant by interrupt masking?
6. Distinguish between the (a) Wait and Stopped states, (b) Running and Operating states, (c) Problem and Supervisor states
7. Describe how the memory protection feature works.
8. What is the difference between the current PSW and an auxiliary PSW?
9. What is a condition code and how is it used in instruction processing?
10. State the PSW changes during a program type of interrupt.
11. What is the difference between an instruction, a command and an order?
12. State the functions of
    (a) the CAW (b) The CCW (c) The CSW.

REFERENCES

1. *IBM System/360 Principles of Operation*, Form A22-684. IBM Corp., White Plains, N.Y.

Chapter 4

Introduction To Assembler Language

Because of intrinsic awkwardness of vocabulary and complete dissociation from "natural" communications, machine language has never been a desirable vehicle for writing programs. Even in very simple machines with small instruction repertoires the tedium associated with specifying instructions as long binary strings makes it very difficult to avoid errors. This is especially true with regard to System/360, since it does present an extensive instruction set, various methods of representing data, and a relatively involved addressing procedure.

Yet despite the availability of an elaborate software structure which offers an unprecedented variety of user-oriented programming languages, there are innumerable occasions which call for very tightly designed procedures written with machine-oriented instructions. The means for doing this are supplied via the Assembler Language. Although far removed from the more "natural" compiler languages, Assembler Language does represent a very powerful programming aid. Its primary asset, of course, is that it gives the programmer direct access to the machine language, allowing him to specify each available instruction in convenient and simplified form. In addition, the language offers him a number of automatically performed services whose equivalents he would have to synthesize himself in machine language.

The software component that mediates between the user's source program

in Assembler Language and the machine's hardware is a very powerful assembler program which does basically the following:

1. It converts Assembler Language instructions to equivalent operations in machine language. When there is a one-to-one correspondence between source and machine language, this merely requires a substitution of operation code. For instructions which are peculiar to the Assembler Language, the assembler program must produce (i.e., generate) the appropriate sequence of machine language instructions.
2. It calculates and assigns numerical addresses to instruction and operand locations, any or all of which can be given convenient names in Assembler Language.
3. The information developed during the above operations is combined to produce a machine language (object) program which, except for a subsequent link-editing operation (performed by the software), is ready for direct loading and execution.
4. While performing the interpretive, conversion, and storage-allocation operations outlined above, the assembler program checks for certain types of errors in the source program (e.g., improper syntax, illegal operation codes, etc.) and produces a display of information about the program. This output may include a listing of the source program, the

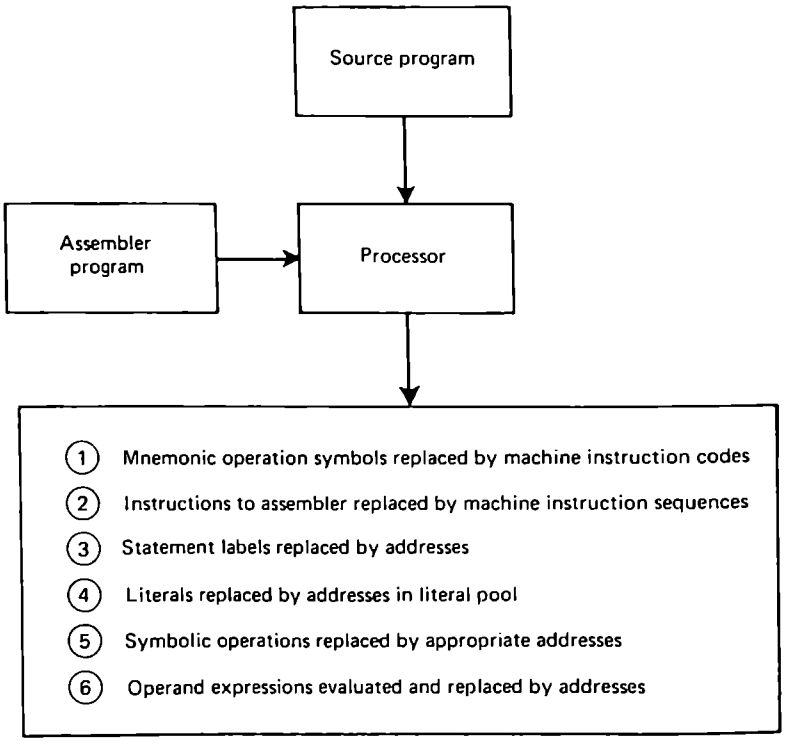

Source program
Assembler program
Processor
1 Mnemonic operation symbols replaced by machine instruction codes
2 Instructions to assembler replaced by machine instruction sequences
3 Statement labels replaced by addresses
4 Literals replaced by addresses in literal pool
5 Symbolic operations replaced by appropriate addresses
6 Operand expressions evaluated and replaced by addresses

Figure 4.1 Summary of Assembler Program's Major Actions

resulting machine-language instructions produced by the assembler program (if it can produce one), a running log of the assembler's activities, and a listing of the errors which may have impeded or prevented the production of an object program.

The action of the assembler is summarized schematically in Figure 4.1. Because of the diversity of possible System/360 configurations, Assembler Language exists in several versions (or dialects). These have been created to make it suitable for the special conditions of particular installations. The versions of Assembler Language are directed specifically toward these variations in memory capacity and are also related to the types and numbers of I/O devices and auxiliary storage units available. Although each programmer will certainly have to be aware of the language features available to him, the basic instruction formats and procedural conventions are common through all versions. We shall present those features that are universal (with the exception of the smallest, most restrictive version), followed by a discussion of those available only in the larger versions (i.e., those functioning under the full Operating System).

4.1 BASIC LANGUAGE COMPONENTS

Before examining the actual programming instructions, we shall establish some of the fundamental properties of the language and see how it is constructed.

Although we have represented a machine-language instruction as a string of hexadecimal digits, it should be clear that this was merely done as a convenience; the only "legitimate" characters available in machine language, of course, are zero and one. This is in direct contrast to the Assembler Language, which allows considerable simplification of vocabulary by making available an extended set of characters.

All versions of the language allow the use of the letters A–Z, the decimal digits 0–9, and the fifteen characters

* + − , ( ) ' . = / # $ @ & blank (b)

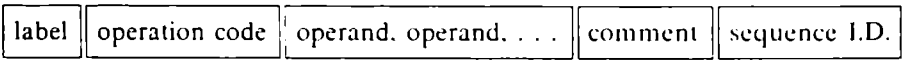
In System/360 context, the "letters" include A–Z, as always, and @, $, and # as well (this is also called the *extended alphabet*). Numeric digits are 0–9 and all others are called special characters. It should be understood that these are the characters accepted by the *language*. That is, no other characters may be used in writing program statements. However, this does *not* mean that these programs cannot accept or work with data which includes any or all of the 256 recognizable characters. (see Chapter 2.)

When viewed in the light of our traditional ideas of a language, Assembler Language is extremely restricted. However, it is truly a language in every organizational sense of the term. As we have seen above, it has its own set

of primitives (allowable characters) from which more complex units (such as words, expressions, and instructions) can be built; it has a grammar in that certain types of components serve different purposes; it also has a definite syntax which defines and restricts the ways in which complex components are synthesized from more basic ones. Having defined the fundamental set of characters from which all language components must alternately be constructed, we shall examine the grammar to determine what some of these components are. Since the integrity of the single command is maintained irrespective of the language being used, we shall use that as our basic framework. To dissociate these commands from direct machine instructions, we shall apply the term *statement* to individual commands in all programming languages except direct machine language.

4.1.1 General Statement Structure

Although the language is extremely versatile, the organization of its components into source statements is relatively simple. We can represent this structure in a general way with a diagram such as shown below:



The label, comment and sequence I.D. are optional, while the other components usually are not. The number of operands varies, of course, with the type of instruction specified in the operation code. When several operands are required, they must be separated by commas.

4.1.2 Statement Labels

In the previous chapter (Sec. 3.3.1) we saw that, once a relative address was selected for the initial instruction in a program, it was up to the programmer to determine or designate adjacent locations for each subsequent instruction, the number of bytes being decided by the instruction length. This is tedious enough for straightforward programs consisting of an unbroken sequence. When branches are involved, the bookkeeping becomes even more complex and the chances for error (e.g., inadvertently branching to the wrong location) multiply. The programmer writing in Assembler Language need not be concerned with such assignments. Once he designates a location for the first instruction (and that procedure is appreciably simplified, as we shall see), he need not specify the address of any statement if his sequence is straightforward. He can do so, of course, if he wishes to earmark certain statements as important points in the program's logical development. (Of course, statements which serve as destinations for branches must be identified.)

An Assembler Language statement is identified by a label of the pro-

grammer's own choosing. The rules for constructing such labels are sufficiently flexible so that the programmer can refer to desired statements by names which are meaningful in the context of the program's logic flow. For example, if a part of his program is devoted to calculating a weight and he wishes to mark the initial statement of that section he can attach a label like WTCALC.

These statement labels have no specific meaning to the assembler program. As part of its regular procedure of examining each source statement, determining the length of its machine-language equivalent, and assigning the proper number of bytes for it, it builds a table called a *symbol table* in which each statement label is matched against the address of the corresponding machine-language instruction generated for it. For example, if the Assembler Language programmer specifies a branch to an instruction labeled WTCALC, one can think of the assembler's activity as consisting of consulting the table to determine the address of the instruction so labeled and placing that address in the operand portion of the corresponding machine-language branch instruction.

In order to construct the symbol table (and for numerous other reasons) the assembler maintains an area in storage called a *location counter*. This is analogous to the instruction counter in the current PSW but it is not a permanently assigned portion of memory, nor is it directly hardware-activated. Instead, it is used during the assembly process of assigning addresses to the instructions produced from the source statements. As each address is assigned, the instruction length is determined and added to it. This sum, representing the address of the next available byte, is stored and updated in the location counter.

Although he doesn't know where the location counter is, the programmer can refer to and control its contents. This is done by specific instructions to the assembler and through the use of a special type of operand.

A statement label may contain from one to eight characters, the first of which must be a member of the extended alphabet. (The basic version of the language has a limit of 6.) The remainder may be alphameric (A, Z, S, $, =, @, or 0–9) but no special characters or embedded blanks can be used. Thus, A4, $LA3SS, J4, W and LOOP4 are legitimate label names, while the labels shown below are not, for the reasons indicated.

Label	*Reason for Rejection*
3A	First character is not alphabetic.
OY VAY	Embedded blank.
AREACODES	Too long.
WT/VOL	Contains a special character.

Each label must be unique in its program (two or more statements may not have the same label), so that only one address is associated with it.

4.1.3 Operation Codes

In place of the binary machine-language operation code the Assembler Language programmer can specify a mnemonic name consisting of a combination of letters which is reminiscent of the operations performed by the corresponding machine-language instruction. For example, the "add halfword" instruction (operation 01001010 or $4A_{16}$) is specified in Assembler Language by operation code AH. These mnemonic codes, though still not quite in "natural" form, are decidedly easier to remember than the machine-language codes.

The mnemonic codes are set up systematically so that their constituent letters relate to the format of the corresponding machine language instruction as well as to the type of operation. This is done by adding one or more standardized letter indicators to a basic operation mnemonic. Table 4.1 summarizes the operation code modifiers. Of course, these modifiers are applied (singly or in combination) only to those instructions where their use would be meaningful.

In addition to this set of operations (called *machine instructions*), which can be considered as being directed to the machine, Assembler Language provides a set of specifiable operations (called *assembler instructions*) for which there are no corresponding single machine-language instructions. Although the programmer indicates them by unique mnemonic codes, just as he does the others, they represent instructions to the assembler rather than the machine. There is no apparent difference to the programmer, but the assembler program must distinguish these codes from those representing single machine-language instructions, since they are processed somewhat

TABLE 4.1 ORGANIZATION OF MNEMONIC MACHINE INSTRUCTION CODES

Added Letter	Meaning	Example
L	Logical	A = add; AL = add logical
C	Character(s)	MV = move; MVC = move characters
H	Halfword	ST = store; STH = store halfword
D	Normalized *long* (floating point)	S = subtract; SD = subtract normalized long
E	Normalized *short* (floating point)	S = subtract; SE = subtract normalized short
W	Unnormalized *long* (floating point)	A = add; AW = add unnormalized long
U	Unnormalized *short* (floating point)	A = add; AU = add unnormalized short
P	Decimal	D = divide; DP = divide decimal
R	RR format	A = add (RX); AR = add (RR)
I	SI format	O = OR (RX); OI = OR (SI)

differently. The machine instructions are introduced in Table 4.2 and the assembler instructions are shown in Table 4.3.

4.1.4 Designation of Operands

Since every machine instruction must contain one or more operands (depending on the particular format), each Assembler Language statement must provide sufficient operand information for the assembler program to be able to provide adequate specifications to the machine. Consequently, the programmer must provide an appropriate number of operands in each statement.

An operand is an expression that can range anywhere from a single simple term to a long and complex string of terms combined by various mathematical operators. We shall first examine the various types of allowable terms and build from that point.

In Chapter 1 we spoke about the fact that System/360 object programs differ from those produced for most machines in that there is no assignment of actual (i.e., absolute) addresses for the various instructions and data fields until immediately prior to execution. We said such programs are relocatable, an attribute that allows great flexibility in control of processing. Where locations are to be assigned just before program execution, operand values in certain program components will change and others will not. This serves as a convenient definition for the two basic types of terms. Those whose values do not change because of the relocation process are called absolute and those whose values do are called *relocatable*.

Operand terms may take on various forms. These can be recognized by the assembler, whose action will be dictated by their structure.

4.1.4.1 Symbols

A symbol, in this context, is a character or string of characters used to represent an address or some value—that is, a name for something rather than that thing itself. Consequently, a symbol is something that the assembler replaces with a value. In this light, a statement label is a symbol, since the assembler ultimately replaces it with an address.

The rule for naming operands (i.e., maximum length, allowable characters, etc.) is the same as those specified previously for statement labels. The processing of these symbolic operands is conceptually the same as described for labels in that the assembler constructs an entry in a table for each unique symbol it encounters, consisting of the symbol accompanied by its designated length in bytes and the assigned address of its leftmost byte. Since these addresses are subject to change, their names are considered to be relocatable terms.

TABLE 4.2 ASSEMBLER LANGUAGE MACHINE INSTRUCTIONS

Mnemonic Code	Machine Language Code (Hex.)	Meaning
A	5A	Add (RX)
AD	6A	Add normalized, long (RX)
ADR	2A	Add normalized, long (RR)
AE	7A	Add normalized, long (RX)
AER	3A	Add normalized, long (RR)
AH	4A	Add halfword (RX)
AL	5E	Add logical (RX)
ALR	1E	Add logical (RR)
AP	FA	Add decimal (SS)
AR	1A	Add (RR)
AU	7E	Add unnormalized, short (RX)
AUR	3E	Add unnormalized, short (RR)
AW	6E	Add unnormalized, long (RX)
AWR	2E	Add unnormalized, long (RR)
BAL	45	Branch and Link (RX)
BALR	05	Branch and Link (RR)
BC	47	Branch on condition (RX)
BCR	07	Branch on condition (RR)
BCT	46	Branch on count (RX)
BCTR	06	Branch on count (RR)
BXH	86	Branch on index high (RS)
BXLE	87	Branch on index low or equal (RS)
C	59	Compare algebraic (RX)
CD	69	Compare long (RX)
CDR	29	Compare long (RR)
CE	79	Compare short (RX)
CER	39	Compare short (RR)
CH	49	Compare halfword (RX)
CL	55	Compare logical (RX)
CLC	D5	Compare logical (SS)
CLI	95	Compare logical (SI)
CLR	15	Compare logical (RR)
CP	F9	Compare decimal (SS)
CR	19	Compare algebraic (RR)
CVB	4F	Convert to binary (RX)
CVD	4E	Convert to decimal (RX)
D	5D	Divide (RX)
DD	6D	Divide long (RX)
DDR	2D	Divide long (RR)
DE	7D	Divide short (RX)
DER	3D	Divide short (R)
DP	FD	Divide Decimal (BS)
DR	1D	Divide (RR)
ED	DE	Edit (SS)
EDMK	DF	Edit and mark (SS)
EX	44	Execute (RX)
HDR	24	Halve long (RR)
HER	34	Halve short (RR)
HIO	9E	Halt I/O (SI)
IC	43	Insert Character (RX)
ISK	09	Insert Storage Key (RR)
L	58	Load (RX)
LA	41	Load Address (RX)
LCDR	23	Load Complement long (RR)
LCER	33	Load Complement short (RR)
LCR	13	Load Complement (RR)

TABLE 4.2 (Continued)

Mnemonic Code	Machine Language Code (Hex.)	Meaning
LD	68	Load long (RX)
LDR	28	Load long (RR)
LE	78	Load short (RX)
LER	38	Load short (RR)
LH	48	Load halfword (RX)
LM	98	Load multiple (RS)
LNDR	21	Load negative long (RR)
LNER	31	Load negative short (RR)
LNR	11	Load negative (RR)
LPDR	20	Load positive long (RR)
LPER	30	Load positive short (RR)
LPR	10	Load positive (RR)
LPSW	82	Load PSW (SI)
LR	18	Load (RR)
LTDR	22	Load and test long (RR)
LTER	32	Load and test short (RR)
LTR	12	Load and test (RR)
M	5C	Multiply (RX)
MD	6C	Multiply long (RX)
MDR	2C	Multiply long (RR)
ME	7C	Multiply short (RX)
MER	3C	Multiply short (RR)
MH	4C	Multiply halfword (RX)
MP	FC	Multiply decimal (SS)
MR	1C	Multiply (RR)
MVC	D2	Move characters (SS)
MVI	92	Move (SI)
MVN	D1	Move numerics (SS)
MVO	F1	Move with offset (SS)
MVZ	D3	Move zones (SS)
N	54	AND logical (RX)
NC	D4	AND logical (SS)
NI	94	AND logical (SI)
NR	14	AND logical (RR)
O	56	OR logical (RX)
OC	D6	OR logical (SS)
OI	96	OR logical (SI)
OR	16	OR logical (RR)
PACK	F2	Pack (SS)
RDD	85	Read direct (SI)
S	5B	Subtract (RX)
SD	6B	Subtract normalized long (RX)
SDR	2B	Subtract normalized long (RR)
SE	7B	Subtract normalized short (RX)
SER	3B	Subtract normalized short (RR)
SH	4B	Subtract halfword (RX)
SIO	9C	Start I/O (SI)
SL	5F	Subtract logical (RX)
SLA	8B	Shift left single algebraic (RS)
SLDA	8F	Shift left double algebraic (RS)
SLDL	8D	Shift left double logical (RS)
SLL	89	Shift left single logical (RS)
SLR	1F	Subtract logical (RR)
SP	F8	Subtract decimal (SS)
SPM	04	Set program (RR)
SR	1B	Subtract (RR)

TABLE 4.2 (Continued)

Mnemonic Code	Machine Language Code (Hex.)	Meaning
SRA	8A	Shift right single algebraic (RS)
SRDA	8E	Shift right double algebraic (RS)
SRDL	8C	Shift right double logical (RS)
SRL	88	Shift right single logical (RS)
SSK	08	Set system key (RR)
SSM	80	Set system (SI)
ST	50	Store (RX)
STC	42	Store character (RX)
STD	60	Store long (RX)
STE	70	Store short (RX)
STH	40	Store halfword (RX)
STM	90	Store multiple (RS)
SU	7F	Store unnormalized short (RX)
SUR	3F	Store unnormalized short (RR)
SVC	0A	Supervisor call (RR)
SW	6F	Subtract unnormalized long (RX)
SWR	2F	Subtract unnormalized long (RR)
TCH	9F	Test channel (SI)
TIO	9D	Test I/O (SI)
TM	91	Test under mask (SI)
TR	DC	Translate (SS)
TRT	DD	Translate and test (SS)
TS	93	Test and set (SI)
UNPK	F3	Unpack (SS)
WRD	84	Write direct (SI)
X	57	Exclusive OR (RX)
XC	D7	Exclusive OR (SS)
XI	97	Exclusive OR (SI)
XR	17	Exclusive OR (RR)
ZAP	F8	Zero and add decimal (SS)

TABLE 4.3 ASSEMBLER INSTRUCTIONS

Mnemonic Code	Identification	Purpose
CCW	Define Channel Command Word	Sets up a location for a CCW and defines its contents.
CNOP	Conditional No-op	An instruction which will fill appropriate bytes with instructions that do nothing but occupy space so that word boundaries can be lined up (see sec. 4.3).
COM*	Define Blank Common Control Section	Reserves a common section of memory which can be consulted by several instruction sequences that have been assembled separately but are being used together as one program.
COPY*	Copy Predefined Coding	Instruction sequences, previously written in Assembler Language and stored externally, can be brought in and incorporated into the program.

TABLE 4.3 (Continued)

Mnemonic Code	Identification	Purpose
CSECT*	Identify Control Section	Assembler Language program can be written in several sections, and each can be assembled separately. The beginning of such a section is identified by this instruction.
DC	Define Constant	Reserves storage for and generates any numeric constant for use in a program.
DROP	Drop Register	Specifies the unavailability of a register previously used as a base register.
DS	Define Storage	Reserves an area of storage under an assigned name.
DSECT*	Identify Dummy Section	Indicates the beginning of an assembled section which is not to be used as part of a program.
EJECT	Start New Page	The assembler causes the instruction following EJECT to appear at the top of a new page in the listing.
END	End Program	Last statement of a program. Stops assembly.
ENTRY	Identify Entry-Point Symbol	Defines a symbol that may be used as a reference by another program.
EQU	Equate Symbol	Assigns designated attribute to a symbol, which can then be used in the program.
EXTRN	Identify External Symbol	Identifies a symbol used in a program but defined in another program (see ENTRY).
ICTL	Input Format Control	Causes a specified alteration in source program format for an entire program.
ISEQ*	Input Sequence Check	Sequence of source statements (if indicated) is checked.
LTORG*	Begin Literal Pool	Defined literals are stored beginning at the location following this statement.
ORG	Set Location Counter	Alters the contents of location counter.
PRINT*	Print Optional Date	Alters the format of printed program listing.
PUNCH*	Punch a Card	Operand is punched on a card.
REPRO*	Reproduce Next Card	The next statement is punched.
SPACE	Space Listing	Inserts blank lines in the listing.
START	Start Assembly	Designates first statement of a program.
TITLE*	Identify Output	Places desired identification at head of assembly listing and punched output.
USING	Use Base Register	Defines availability of register for base register use (see DROP).

*Not available with Basic Assembler.

For example, the instruction

```
CALC  AP    VAR1,VAR2
```

breaks down as follows:

1. CALC is a symbol used for the statement label. Using its location counter, the assembler will assign an address for this statement and store it in a symbol table, together with the label.
2. AP is the mnemonic operation code representing the Add Decimal instruction (machine code 11111010). This is an SS type instruction requiring two operands representing storage addresses.
3. VAR1 and VAR2 are symbols representing the two operands to be added together. As in the case of the statement label, these names are stored in a table together with corresponding addresses.

The assumption here, of course, is that the programmer has previously defined the contents of VAR1 and VAR2. Thus, he can now call for their addition without concerning himself with their actual locations.

4.1.4.2 Self-Defined Terms

In contrast to symbols, self-defining terms are themselves the actual values used in the program. When the assembler encounters something it "recognizes" as a self-defining term, it does not assign a value to it. Instead, it builds it right into the resulting machine-language instruction. For instance, the statement

```
SUM1  AR    5,7
```

which carries a label SUM1 and specifies the addition of the contents of general registers 5 and 7, would be converted to

```
0001101000101111
```

Since self-defining terms are not changed by program relocation, they are considered to be absolute terms.

The assembler accepts four types of self-defining terms:

1. *Decimal constants.* A decimal self-defining term is any unsigned decimal integer up to eight digits long. Its value may not exceed 16,777,215 ($2^{24}-1$). (Of course, if it is used to specify a general register, its value must be one of the numbers 0–15.)
2. *Hexadecimal self-defining terms.* These are unsigned hexadecimal integers up to six hexadecimal digits in length. In order to avoid ambiguity as far as the assembler is concerned, these constants must be written with an apostrophe immediately before and after the value. The first apostrophe is immediately preceded by the letter x. Thus the hexa-

decimal value C4A is written as a self-defining term in the form X'C4A'. No hexadecimal constant may exceed the value X'FFFFFF'.

3. *Character self-defining terms.* This type of constant, which may be from one to three characters long, may consist of a combination of any of the 256 characters recognized by System/360. Character constants are enclosed by apostrophes and immediately preceded by the letter C. Thus, for example, a single blank is expressed in the form C' '. In order to keep things straight for the assembler, special handling must be used when a character constant is to include an apostrophe or an ampersand (&). When this is the case, it is necessary to specify two apostrophes or ampersands for each apostrophe or ampersand to be included in the constant. Thus, for example, when the assembler encounters a specification like C'W''5' it will make arrangements to use W'5. Similarly, the specified constant C'''/&&' will generate the value '/&. If it were to encounter something like C'I'M' it will interpret this as an error and produce a message to that effect. The most restricted form of the Assembler Language (we shall call it the Basic Assembler) restricts the length of such constants to a single character. (The necessity for double apostrophes and ampersands still holds.)
4. *Binary self-defining terms.* This type of constant (unavailable in the Basic Assembler) is written as an unsigned binary integer (i.e., a string of 1's and 0's) enclosed by single apostrophes and immediately preceded by the letter B. Such constants, whose maximum length may be 24 bits, are used most frequently in comparison intructions. For instance, the Test Mask instruction (mnemonic code TM) allows a bit-by-bit comparison between a designated storage location and a binary constant. Thus the statement

```
TM   BITBIT,B'10111010'
```

specifies that the first bit in the location specified by the symbol BITBIT is to be compared with the first bit of the constant B'10111010', and so on.

Since self-defining terms are values which do not change when a program is relocated; they are considered to be absolute terms.

4.1.4.3 Literals

A literal is a value that is used by a program as data. When specified as an operand in an Assembler Language instruction, it is written as a constant (using one of the forms described in the previous section) immediately preceded by the character =. There are additional types of literals, as we shall see in Sec. 4.3, but they all follow this same general format.

Although the literal appears to be like a constant, its treatment by the assembler is quite different. We said that the assembler built the value of a constant right into the instruction in which it appears. In contrast, when it encounters a literal, it generates the indicated value and stores it together with other literals in a section of memory reserved for that purpose. This area is called a *literal pool*. Having done this, the *address* of the literal, rather than the value itself, is built into the resulting machine-language instruction. For example, if the RX type Load operation (mnemonic code L) were to appear in the statement

```
L   4,=X'5E'
```

it would represent a request for the program to load the hexadecimal constant 5E (the binary string 01011110) into general register 4. However, when the assembler converts this to a machine-language instruction, the second operand will not be the constant value. Instead, it will be the address of the byte in the literal pool containing that constant.

Because of these special properties, the use of literals is fairly restricted. Specifically, only one literal may appear in a machine instruction statement. Furthermore, a literal cannot be combined with any other term. Since literals are processed in such a way that their addresses, rather than their values are specified in machine-language instructions, they must be considered as relocatable terms. They are not recognized, and therefore cannot be used, with the Basic Assembler.

4.1.4.4 Symbol Length Specification

There are occasions where it is necessary to specify the length of an operand in addition to or instead of its address or value. The most common instance occurs in instructions of the SS format, in which the programmer can specify a field length for each of the two operands. If he knows what these lengths are, he can specify them as appropriate constants. In addition, the Assembler Language allows him to specify a length he doesn't know and have the assembler determine what it is. This is done by using the name (symbol) of the field in whose length he is interested, immediately preceded by the two characters L'. To illustrate, let us suppose that the field name VAR1 is two bytes long and that VAR2 represents a field with a length of four bytes. The SS type instruction Move Characters (mnemonic MVC), when used in the statement

```
MVC   VAR2(L'VAR1),VAR1
```

causes the two bytes contained in VAR1 to be duplicated in the first two bytes of VAR2.

Since a field length does not change when a program is relocated, this type of term is considered to be absolute. The Basic Assembler cannot handle such terms.

4.1.4.5 Location Counter Reference

The Assembler Language allows the programmer to make direct reference to the location counter by using an asterisk (*) as an operand. The asterisk refers to the address of the first byte of the instruction being worked on at that instant. For example, if we were to use the MVC instruction (see previous section) in the statement

```
IDENT   MVC   *,VAR1
```

we would be saying, in essence, "reproduce the contents of the byte located at the address corresponding to the symbol VAR1 in the first byte of this instruction (i.e., the instruction labeled IDENT), destroying the previous contents there." In other words, this is the same as saying

```
IDENT   MVC   IDENT,VAR1
```

The location counter reference, of course, is a relocatable term.

Table 4.4 summarizes these terms according to their absolute or relocatable attribute.

TABLE 4.4 LOCATION ATTRIBUTE OF OPERAND TERMS

Type of Operand Term	Location Attribute
Symbol	Relocatable
Self-Defining Term	Absolute
Literal	Relocatable
Symbol Length Specification	Absolute
*	Relocatable

4.1.4.6 Combination of Terms to Form Operand Expressions

The assembler's services extend well beyond the replacement of symbolic operands with their corresponding addresses. The programmer may construct an arithmetic expression consisting of various acceptable combinations of terms and the assembler will do the necessary calculations and substitutions to reduce their expression to a single value which will be used as the address.

As in most other contexts, an expression consists of a series of terms connected by operators. The operators available for this purpose are + (addition), − (subtraction), * (multiplication), and / (division). The division operator is not available in the Basic Assembler. An expression may contain from one to three terms (up to 16 in the larger assembler version). These combinations may include symbols, constants, length references and location-counter references in accordance with a set of specifically defined rules:

1. An expression cannot contain two adjacent terms or operators. The sequence must always be term, operator, term, operator, etc.

2. An expression cannot begin with an operator. Thus, −3+VAR1 is not allowed, but VAR−3 is acceptable.
3. A literal cannot be combined with additional terms.

When the assembler program encounters an operand expression consisting of more than one term, it reduces that expression to a single value in accordance with the following sequence:

1. Each term is replaced by its value. That is, symbols looked up in the symbol table and their corresponding numerical addresses are placed in the expression; constants are converted to their binary equivalents. A literal (which is a one-term expression) is replaced by the corresponding address in the literal pool.
2. The indicated arithmetic is performed, with the assembler working from left to right. First, all multiplications and divisions are done. Then the expression is scanned again to find and perform the additions and subtractions. Thus, for example, the expression VAR1−3*VAR2 is evaluated as VAR1−(3*VAR2) rather than (VAR1−3)*VAR2.
3. When a division is indicated, the results will always be an integer—all remainders are truncated. Thus the operation 7/10 produces a result of zero.
4. The priorities given in Rule 2 above may be altered by enclosing selected terms in parentheses. For example, the expression VAR1−3*VAR2 will be handled by doing the indicated multiplication first and then subtracting the product from VAR1. By adding parentheses and writing the expression as (VAR1−3)*VAR2, we force the assembler to reduce the contents of the parentheses to a single value before it does anything else. Parentheses may not be used for this purpose in the Basic Assembler. Up to five sets of parentheses may be used in an expression in the most extensive version of the assembler, while only one set is allowed in all other versions.

Absolute and relocatable terms may be combined to form legitimate operand expressions as long as the following rules are obeyed:

1. A relocatable term may not be used with the * or / operators.
2. An absolute expression (i.e., an expression which works down to an absolute value) may contain relocatable terms either with or without absolute terms. When this happens, the relocatable terms must appear in pairs, with the terms in each pair having opposite signs. Thus if VAR1 and VAR2 are relocatable symbolic terms, the operand expression 3+VAR1−VAR2 is legitimate, but the expression 3+VAR1+VAR2 is not. Since the difference between two relocatable terms is an absolute value, an expression like 7*(VAR1−VAR2) is valid while 7*VAR1−VAR2 is not.

3. A relocatable expression (one which reduces to a relocatable value) may contain absolute and relocatable terms in combination. Irrespective of the use of absolute terms, there must be an odd number of relocatable terms.
4. If an expression contains three relocatable terms, two of them must be paired (as explained in 2 above) and the other one may not be directly preceded by the − operator.

Table 4.5 shows some examples of absolute and relocatable operand expressions.

TABLE 4.5 OPERAND EXPRESSIONS (VAR1, VAR2 and VAR3 are symbols)

Expression	Type	Legal Status
=X'3B'	Relocatable	Valid
*+VAR1	—	Illegal—unpaired relocatable terms
*−VAR1	Relocatable	Valid
3*L'VAR2/2	Absolute	Valid
2*VAR1+VAR2−VAR3	Relocatable	Valid
4*VAR2+15	—	Illegal—relocatable terms used with*
=12+VAR2−VAR3	—	Illegal—use of literal in multiple-term expression
2+L'VAR1*(VAR2−VAR1)	Absolute	Valid

4.1.5 Comments

Assembler Language programmers can intersperse their source programs with any sort of narrative information. This usually takes the form of comments concerning the flow of logic or an explanation of the operations performed at a given point. Such descriptive material may contain any member of the acceptable character set (as defined at the beginning of Sec. 4.1). The assembler does not process comments. It merely causes them to be printed as part of the listing.

4.1.6 Identification-Sequence Field

Provisions are included which allow the programmer to supplement each statement with up to eight characters that identify the program and provide sequencing information about the accompanying statement. As is the case with comments, this information, if included by the programmer, is reproduced as part of the listing. Also, the programmer may call for the assembler to examine this field by means of the ISEQ assembler instruction.

4.2 CODING CONVENTIONS

Although other input media can be used, we shall orient our discussion around the punched card as the standard vehicle for submitting source programs and data unless otherwise indicated. Furthermore, we shall avoid undue complications by adhering to a standard practice of using a separate card for each statement.

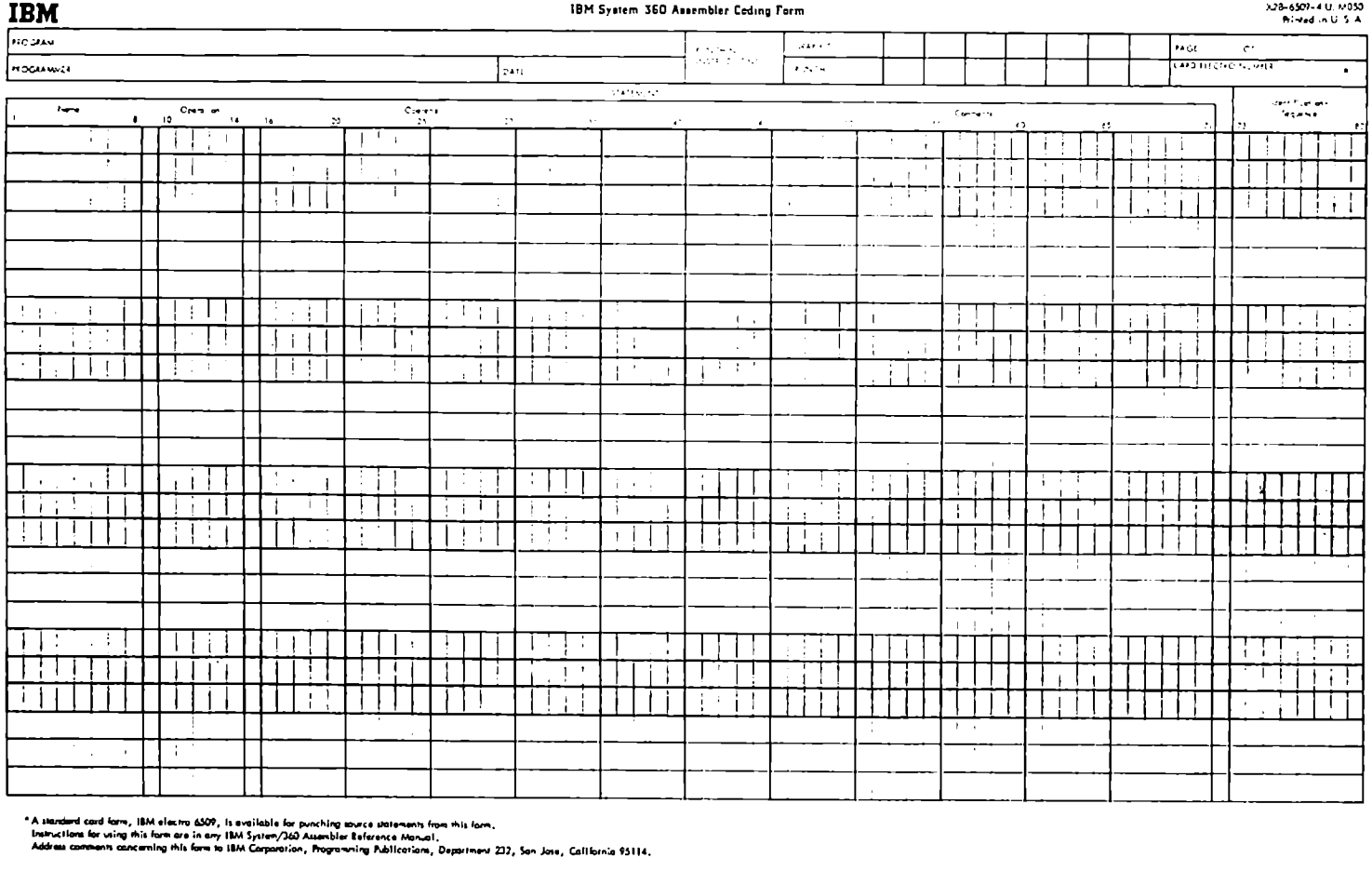
4.2.1 Coding Form

As a convenience to programmers and keypunch operators, a standard coding form has been designed (Figure 4.2) in which each line accommodates a single Assembler Language statement and corresponds to a full 80-column card. Although there are certain guidelines shown on the coding form, the format is relatively unrestricted with respect to length and placement of statement components. The statement itself (everything excluding the identification-sequencing field) is placed between columns 1–71. Consequently, these boundaries are known as the *begin* and *end* columns. 73–80 are reserved for identification-sequencing information whose inclusion, of course, is optional. These boundaries can be changed by means of the ICTL assembler instruction. If a statement is too long to fit in the 71 available columns, it can be continued on a second card (not in the Basic Assembler). This is done by specifying any character other than a blank in column 72 and continuing on the next card starting in column 16 and going through 71. The character in column 72 of the first card is not treated by the assembler as part of the statement and the first 15 columns of the second card (called the *continuation card*) are ignored. The most extensive version of the assembler allows two such continuation cards.

4.2.2 Order of Statement Components

As seen in Figure 4.2, the coding form is ruled to provide for an eight-character label, a five-character operation code and a 56-character field for the operand(s) and comments. These boundaries may be ignored, as long as the rules given below are followed:

1. The sequence must always be: label (if used), operation code, operand list, comments (if used).
2. Each of the above-mentioned components must be separated from its neighbors by at least one blank.
3. Columns 72–80 cannot be used for part of the statement (without an ICTL change, that is) even though no sequence information is included.



* A standard card form, IBM electro 6509, is available for punching source statements from this form.
Instructions for using this form are in any IBM System/360 Assembler Reference Manual.
Address comments concerning this form to IBM Corporation, Programming Publications, Department 232, San Jose, California 95114.

Figure 4.2 IBM System/360 Assembler Coding Form

4. When a statement label is used, it must always start in the begin column, regardless of its length.
5. If a continuation card is used, the label and operation code, plus at least one blank following the code, must be given on the first card.
6. When more than one operand is used, the operands must be separated by commas, with no blanks between the last character of the operand and the comma following it.

Comments may be entered in two ways. First, they may follow the last operand (with at least one blank in between) and extend to the end column. In addition, an entire line (or card) up to column 71 may be used for comments by putting an asterisk in the begin column. The entire contents of columns 2-71 will then be treated as a comment.

4.2.3 Conventions for Handwritten Statements

To help minimize ambiguities with respect to keypunching program statements, the following conventions have been informally set up and are in general use:

1. All letters in programs are capitalized, except where a piece of data or a constant specifically calls for a lower case letter.
2. The letter between N and P in the alphabet is written Ø to distinguish it from zero.
3. The letter after Y is written Ƶ to distinguish it from 2.
4. The number is 1; the letter is I.
5. When the programmer wishes to indicate the specific presence of blanks, he writes ƀ for each one.

4.3 RESERVATION OF STORAGE

To round out this overall examination of Assembler Language we shall take a close look at one of the language's most powerful facilities. By means of a relatively simple set of assembler instructions, the programmer is able to assign a symbolic name to a desired amount of memory. Furthermore, he can bring about the internal generation of any type of constant value and have it stored in memory under a name of the programmer's choosing. These filled and unfilled storage areas, once defined, are then available for use as symbol terms for operands.

4.3.1 The DC Instruction

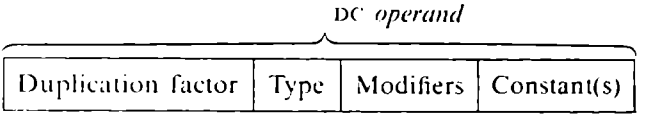
Data values are internally generated and stored by means of the DC (Define Constant) instruction. The general format is as follows:

label DC *operand*

For the most extensive assembler, the instruction may take more than one operand, with adjacent operands being separated by commas. No blanks may appear between the last character of an operand and the separating comma.

An operand can be composed of as many as four components called *subfields*. These are not terms, in the sense discussed in the previous section. The subfields have specific functions and their sequence is prescribed.

A DC operand, in its most complete form, contains the following subfields in the sequence shown:



No blanks may occur between subfields and there can be no blanks within a subfield unless they are part of a character constant. An operand need not contain all of the subfields.

We shall discuss the structure and uses of the subfields in a different order than that of their required sequencing. This will permit the buildup of the complex operand from simpler forms.

4.3.1.1 The "Constant" and "Type" Subfields

The fourth subfield specifies the actual constant that is to be stored. In this context, we can divide the possible types of constants into two basic classes: address constants and data constants. The former category includes four types which represent address values; the others indicate numerical or character values that represent items meaningful to the programmer.

A data constant is specified by enclosing it in single quote marks (apostrophes). Parentheses are used in the same way for address constants. More than one constant may be specified in the same subfield (except in the Basic Assembler) but they must all be of the same type. When such a multiple specification is made, the constants are separated by commas (no blanks) and the whole group is enclosed by the appropriate brackets. Thus, for example, a series of four decimal integers would be written

'14,275,6,30988'.

Certain types of constants may be specified only one to an operand, even in the advanced assembler. These will be identified as we discuss the individual types. Each type of constant is accompanied by a unique type identifying code, which appears in the second subfield of the operand. Thus the fourth subfield tells the assembler what value to store and the second subfield indicates the form to be used. All of these constants may be used as literals, except as indicated.

1. *Character Constant.* This type, denoted by a C in the second subfield, can be a string of up to 256 characters in length. Any of the 256 recognizable character types may be used. The rule for specifying ' and & (Sec. 4.1.3.2) applies here. No version of the assembler allows more than one character constant in an operand. Thus the statement

```
IDENT  DC    C'MOISH''S'
```

   will reserve seven bytes containing MOISH'S, with the first byte having the name IDENT.

2. *Fixed-Point Constants.* This type consists of a signed or unsigned decimal number followed by an optional exponent. The number may be an integer (no decimal point indicated), a fraction (decimal point precedes the first digit) or a mixed number (decimal point between two of the digits). If either of the latter two types of values is indicated, then a scale modifier must be specified also (see Sec. 4.3.1.2). Otherwise the fractional portion will be lost. If an exponent is used, it is written immediately after the number. The form is E*p* where *p* is a decimal number (signed or unsigned) indicating the power of ten to which the number is to be raised by the assembler to obtain the desired magnitude. Thus, for example, the fixed-point constant '−31E−3' will cause the assembler to generate the binary equivalent of $-31X10^{-3}$ or −.031. Allowable exponent range is −85 to +75. There are two type codes for fixed-point constants. An F indicates that the constant should be stored as a fullword while an H directs the assembler to use a halfword, with the value occupying as many of the rightmost bits as required. Greater lengths may be specified by proper use of the third subfield, as we shall see. If the specified number is too large for the indicated field length, the necessary leftmost bits are truncated.

   To illustrate, let us look at the statement

```
VALI   DC    F'614E1'
```

   As a result of this statement, the assembler will allocate a fullword of storage, with the name VALI assigned to its first byte. In that fullword will be placed the binary equivalent of 6140.

3. *Floating-Point Constants.* If it is desired to store a numerical value in floating-point form, the number is specified exactly as described for fixed-point constants (see previous section). A code of E in subfield 2 reserves a fullword, while a D reserves a doubleword. Thus the statement

```
FLTVAL  DC    D'4271E-2'
```

   will cause a value of +42.71 to be stored as a floating-point doubleword whose first byte is located by the symbol FLTVAL.

4. *Decimal Constants.* These are not fixed-point constants, even though they are written as decimal numbers in the DC statement. The presence or absence of a decimal point makes no difference to the assembler, since this type of constant produces a value which is stored without conversion to binary form. Thus its use in the statement is merely a convenient reference. If a P is specified in subfield 2, the value is stored in packed decimal format (two decimal digits to a byte, with the rightmost digit and sign occupying the rightmost byte). An extra half-byte of zeros is added automatically if needed.

   A Z in subfield 2 will cause the value to be stored in zoned format (one byte per digit, with the rightmost byte containing the sign and the rightmost digit). In both P and Z types, the assembler assigns exactly the number of bytes needed. This can be altered by an appropriate specification in subfield 3. Neither of these types of constants is recognized by the Basic Assembler. The statement

```
PACKEM  DC    P'32.7'
```

   will reserve two bytes for the value 327+, with the first byte having the symbol PACKEM assigned to it. The decimal point serves as a guide to the programmer for future alignment action.

5. *Hexadecimal Constants.* A string of up to 512 hexadecimal digits can be generated in a DC statement. Each pair of hexadecimal digits is stored in a byte and the symbol is assigned to the leftmost byte. The subfield 2 code is X. All assembler versions prohibit the specification of more than one hexadecimal constant per operand.

   Use of an eight-digit hexadecimal constant makes a convenient way to specify the contents of a binary word. For example, the statement

```
BINWRD  DC    X'FF0000FF'
```

   sets up a fullword with contents 11111111000000000000000011111111, with the first byte having the name BINWRD.

6. *Binary Constants.* A combination of up to 256 1's and 0's may be specified by writing a B in subfield 2 and the desired string enclosed in apostrophes. The assembler will pad to the left with zeros to make it come out to the nearest integral number of bytes. Thus the statement

```
BNSTRG  DC    B'10110010101'
```

   will produce

```
BNSTRG BNSTRG+1
 00000101 10010101
```

   Only one binary constant may appear in a DC operand. The basic assembler prohibits their use here altogether.

7. A-*type Address Constant.* This type of constant can be represented by any legitimate absolute or relocatable expression as defined in Sec.

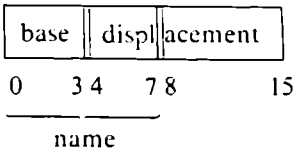
4.1.3.6. A code of A is used in subfield 2. The assembler performs the calculations indicated in the expression and stores the result in the rightmost portion of a four-byte field. This automatically assigned length can be altered in subfield 3. To illustrate, let us look at the statement

```
WHERE   DC      A(* + 208)
```

As a result of the assembler's action, a four-byte field whose first byte carries the name WHERE will contain the address represented by *+208 (in other words, WHERE+208).

In addition to these types of expressions, an A-type address constant may be a *complex relocatable expression*, in which unpaired relocatable terms are allowed. This is beyond our scope here, but pertinent information can be found in IBM's manuals dealing with the Assembler Language.[1-3]

8. Y-*type Address Constants.* By specifying a Y instead of an A in subfield 2, the address constant given in subfield 4 will be stored in two bytes rather than four. This length can be reduced but not exceeded by an additional specification in subfield 3. The Y-type address constant is not accepted by the Basic Assembler and should not be used in any program to be executed under the full Operating System on any processor larger than 32K.
9. S-*type Address Constants.* Addresses may be stored in base-displacement form by specifying an S in subfield 2 and indicating the desired address (enclosed in parentheses) as a legitimate absolute or relocatable expression. The assembler will evaluate the expression, and perform the necessary manipulation to express the result as a base and displacement, said result being stored in a halfword as



An S-type address constant can also be expressed directly in base-displacement form by writing two absolute expressions as follows:

```
(expl(exp2))
```

The assembler will use the first of these as a displacement and the second as a base. Thus, if VAR1, VAR2, and VAR3 are relocatable terms, the statement

```
PLACE   DC      S(VAR2-VAR1+VAR3)
```

will cause the resulting address to be connected to base-displacement

form (the assembler automatically uses the proper register) and to be stored in the halfword beginning at PLACE. On the other hand, the statement

```
PLACE1  DC      S(VAR2-VAR1(12))
```

will store an address having a base of 12 and a displacement represented by VAR2−VAR1 to be stored in the halfword beginning at PLACE1. s-type constants cannot be used as literals and are not recognized by the Basic Assembler.

10. v-*type Address Constant.* A v in subfield 2 indicates that the address constant in a subfield refers to the address of a field that is defined in a separately assembled program and is to be used as a branch destination. We shall not be concerned with it here.

4.3.1.2 Boundary Alignment

In order to gain a clear understanding of a number of subsequent topics, we shall have to digress briefly to discuss an aspect of storage assignment which must often be employed to assure program compatability among the various System/360 processors.

As pointed out in Chapter 1 (Table 1.3), various processor models access different numbers of bytes during a machine cycle to exploit their internal speed capabilities more fully. The implication of this variation is that special care must be taken in the storage of instructions and data. If this is not done, the use of a program on a machine with access properties differing from the one for which the program was originally written can easily upset proper execution because of the use of improper-length data fields and the like.

The way this anomaly is circumvented is to set up and use standardized storage *boundaries*, so that software and problem programs can anticipate the beginnings of separate fields. For this purpose the following rules have been established.

1. All halfwords must begin at an address that is a multiple of two. That is, the rightmost hexadecimal digit of the address must be 0, 2, 4, 6, 8, A, C, or E.
2. All fullwords must begin at an address that is a multiple of four (right hexadecimal digit of 0, 4, 8, or C).
3. All doublewords must begin at an address that is a multiple of eight (right hexadecimal digit of 0 or 8). In most cases the assembler will handle such alignment automatically by using the location counter. When alignment is necessary, the appropriate number of "empty" bytes is added. When such alignment is necessary or desired in a sequence

of instructions, the programmer can induce it by using the CNOP assembler instructions. Its general form is

```
CNOP    abs1,abs2
```

where *abs*1 and *abs*2 are absolute expressions. This changes the location counter value (if the assembler finds it needs changing) to a halfword, fullword or doubleword boundary. *abs*1, which can be either 0, 2, 4, or 6, specifies at which byte number the location counter is to be aligned; *abs*2, which is either 4 or 8, indicates whether that byte number is in a word or doubleword. If adjustment is not required, nothing happens. If it is, the assembler effects it by inserting the necessary number of two-byte no-op (do nothing) instructions which fill the space without disrupting the sequence. Note that no label is attached.

4.3.1.3 The "Modifier" Subfield

We have seen that the specification of a constant type in subfield 2 and the indication of its value in subfield 4, without additional information, are sufficient to arrange for the reservation of storage and generation of the desired value. The length assigned in such cases is obtained by the assembler either by examining the constant (as is done with C-type constants) or assigning a default length that is set in the assembler's design. (This is known as an *implied length*.) These, and other assumed values, can be overridden by appropriate specifications in subfield 3. More than one type of modifier may appear in the same subfield.

1. *Length Modifier.* The number of bytes to be allocated for a particular constant can be explicitly designated in a DC statement by the subfield L*n*, where *n* is an unsigned decimal number, or an absolute expression in parentheses. Of course, any symbol used in this expression must have been previously defined. When used within the limits set for each type of constant (given in Table 4.6), the length modifier overrides the implied length and dispenses with automatic boundary alignment: if a limit is exceeded, the allowable maximum is used. If the constant requires more room than specified in the length modifier, it is truncated as shown in Table 4.6. Similarly, it is padded with blanks or zeros if more than sufficient length is specified. Thus the statement

```
BERL    DC      CL3'CAPE'
```

will reserve three bytes (first one named BERL) which will contain CAP, while the statement

```
SHMERL  DC      CL5'CAPE'
```

TABLE 4.6 SUMMARY OF CHARACTERISTICS OF CONSTANTS

Type	Code	How Specified	Implied Length	Alignment	Allowable Length	Exponent Range
Character[a]	C	Characters	As required	Byte	1-256 bytes[b]	—
Fixed Point	F	Decimal digits	4 bytes	Word	1-8 bytes	−85 to +75
Fixed Point	H	Decimal digits	2 bytes	Halfword	1-8 bytes	−85 to +75
Floating Point	E	Decimal digits	4 bytes	Word	1-8 bytes[b]	−85 to +75
Floating Point	D	Decimal digits	8 bytes	Doubleword	1-8 bytes[b]	−85 to +75
Decimal (packed)[+]	P	Decimal digits	As required	Byte	1-16 bytes	—
Decimal (zoned)[+]	Z	Decimal digits	As required	Byte	1-16 bytes	—
Hexadecimal[a]	X	Hexadecimal digits	As required	Byte	1-256 bytes	—
Binary[a+]	B	Binary digits	As required	Byte	1-256 bytes	—
Address[0]	A	Any expression	4 bytes	Word	1-4 bytes	—
Address[0+]	Y	Any expression	2 bytes	Halfword	1-2 bytes	—
Address[0+]	S	One absolute or relocatable expression, or two absolute expressions in the form (C1(C2))	2 bytes	Halfword	2 bytes	—
Address[0+]	V	Relocatable symbol	4 bytes	word	3 or 4 bytes	—

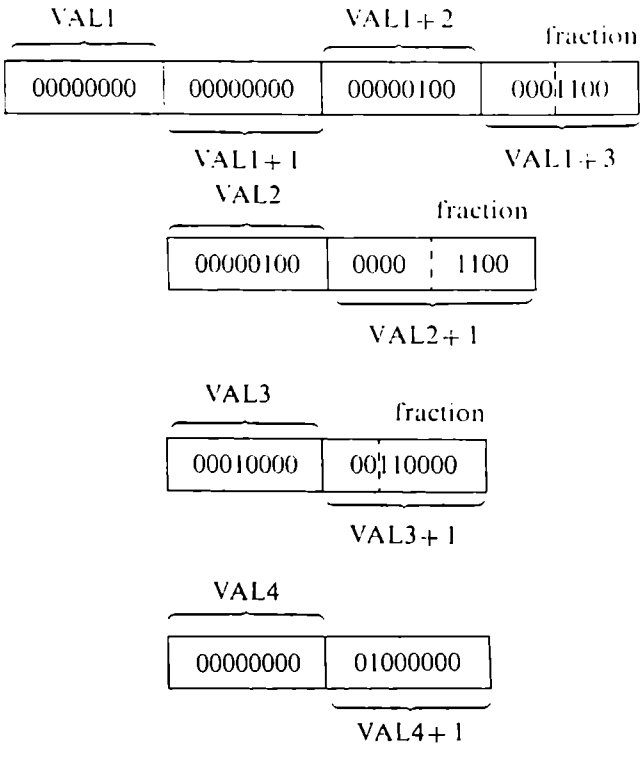
[a] Only one constant per operand.
[b] Truncation or padding occurs on the right; for all others it is on the left.
[0] Multiple constants for operand only in advanced assemblers.
[+] Not available with Basic Assembler.

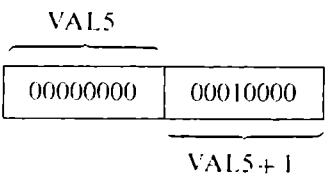
will reserve five bytes (first one named SHMERL) containing CAPE b. Length modifiers on H, E and D type constants will not be accepted by the Basic Assembler. Also, the modifier cannot be an expression.

2. *Scale Modifier.* This modifier is denoted by S*n*, where *n* has the same meaning as in the length modifier. When applied (it may be used only with F, H, E, and D type constants) it sets the amount of internal scaling (magnitude adjustment) for that constant. If a fixed-point (F or H) constant includes a fractional portion, a scale modifier must be specified to prevent loss of the fractional part. A *positive* modifier value specifies the number of bits at the right-hand end of the field to be used for the fractional part. A *negative* modifier expresses the number of bits to be deleted from the right-hand end of the integer portion. Otherwise stated, a positive value shifts the binary point that many places to the left, while a negative value shifts it to the right. For example, the statements

```
VAL1  DC   FS4'64.75'
VAL2  DC   HS4'64.75'
VAL3  DC   HS6'64.75'
VAL4  DC   H'64.75'
VAL5  DC   HS-2'64.75'
```

produce stored constants as shown:





When used with a floating-point (E or D) constant, the scale modifier must be positive. It indicates the number of *hexadecimal* positions that the fraction is to be shifted to the right. Since the initial conversion process includes normalization (adjustment of the exponent so that the fraction does not have a zero as its leftmost digit), the scale modifier will introduce one or more zeros there. (Shifting the fraction to the right, which is what actually happens, is equivalent to shifting the point to the left.) Consequently, the exponent is readjusted before final storage. Thus the constant in the statement

```
CON1   DC     E'32.174E-3'
```

which represents a value of .032174, will be normalized and stored (conceptually) as $.32174 \times 10^{-1}$. If we added a scale modifier to make it read

```
CON1   DC     E5'32.174E-3'
```

the assembler will store the equivalent of $.032174 \times 10^{1}$. Scale modifiers cannot be used with the Basic Assembler.

There are two additional types of modifiers which will only be mentioned here. The exponent modifier, written E*n*, (not available in the Basic Assembler) allows the exponent in an F, H, D, or E type constant to be adjusted *over and above* that given in the E*n* specification that goes with subfield 4. It is used to provide a common adjustment to *all* of the constants in a multiple constant subfield, where each constant may or may not have its own individual E*n* specification as well. The second type, available only in the advanced assembler, allows the programmer to specify the length of a constant in bits.

4.3.1.4 The "Duplication Factor" Subfield

The constant type designation in a DC statement may be preceded by an unsigned decimal constant or an absolute expression, the latter enclosed in parentheses. If this subfield is present, the assembler will store the specified constant as many times as is indicated. The duplication factor can be zero, with the effects as described in Sec. 4.3.2. For instance, the statement

```
LOTS   DC     3CL4'H00'
```

will reserve 12 bytes (first one named LOTS) which will contain

HOObHOObHOOb

A-type constants cannot be duplicated in the Basic Assembler and S-types cannot be duplicated in any but the most extensive assembler.

4.3.2 The DS (Define Storage) Instruction

The assembler can be instructed to assign names to designated amounts of storage without any particular contents being specified. This is done with the DS statement whose general form is

label DS *operand*

The DS operand has exactly the same construction as that of the DC statement, with the following exceptions:

1. Subfield 4 (the actual constant) need not be specified.
2. Areas intended for storing characters or hexadecimals may be up to 65K bytes long for the most extensive version of the assembler (256 bytes otherwise).

   The assembler will reserve the amount of storage specified in subfield 3. Note that if there is a constant given in subfield 4 and no length in subfield 3, the assembler will use that constant to determine how much storage to reserve. It will *not* generate that constant. This provides the programmer with the convenience of not having to figure out the length of the field he wants if he knows what its ultimate contents will look like. Alignment is performed as in the DC statement. Since this statement stores no information, subfield 2 really serves to control the basic amount of storage reserved, as well as the type of alignment performed. As with the DC, use of subfield 3 cancels automatic alignment. Implied lengths and alignment action are those specified for the DC statement (Table 4.6) for most constant types. C, X, P, and Z constants automatically cause the reservation of only one byte. Consequently, these require a length modifier or a fourth subfield if more room is desired.

   The duplication factor is used as in the DC statement. Thus

   FIELD DS 3CL5

   reserves three five-byte areas starting at FIELD, FIELD+4 and FIELD+9. This is not the same as saying

   FIELD DS CL15

   since the latter reserves only one area, even though its length equals that of the three in the previous statement. This distinction is most important, of course, in SS type instructions.

It is important to reemphasize the fact that nothing but storage assignment takes place. Consequently the programmer cannot count on reserved storage containing zeros, or blanks, or anything else.

4.3.2.1 Forcing Alignment

The programmer can force the location counter to be reset to a desired boundary in the midst of a series of DS or DC statements by writing a DS statement with the appropriate field type (D, F, or H) and a duplication factor of zero. Having done this, he can follow with his desired DS. Thus the sequence

```
         DS    0D
PLACE    DS    CL6
```

would set the location counter to the next double word boundary and would reserve the next six bytes beginning at that boundary.

4.3.2.2 Reserving Areas for Data Collections

The zero duplication factor can also be used to assign a name to an area without actually reserving storage. Then the area can be reserved as several separate fields, each with its own name, and all of them can be referred to collectively by the single area name. For instance, the sequence

```
ADDRESS  DS    0CL35
STRNUM   DS    CL5
STRNAM   DS    CL15
CITY     DS    CL15
```

will reserve 5 bytes under the name STRNUM, and 15 each under the respective names STRNAM and CITY. The name ADDRESS refers to all 35 bytes. In this way the programmer can read or write the entire address with one statement when appropriate and he can manipulate each component separately at other times.

4.3.3 The EQU Instruction

This assembler instruction does not actually reserve or allocate storage. However, its use is often closely allied with DC and DS. By using the instruction it is possible to assign a name to a particular address or other value. Then the assembler will substitute the address or value whenever that name is encountered. The value thus symbolized may or may not have another name already. EQU statements take the general form

```
name   EQU    operand expression
```

For example, the statement

```
REG6  EQU   6
```

sets up a mechanism for the automatic replacement of the symbol REG6 with a value equivalent to 6_{10}. Then, if General Register 6 is to be used in a statement, it can be referred to by specifying REG6. (Of course, REG6 can be used anywhere a 6 is required, but in this example the name was selected specifically because it was intended to represent a register.)

If VAR1 is an address symbol that was previously defined, the statement

```
DISPLY  EQU   VAR1+4*7
```

assigns the name DISPLY to the location 28 bytes after VAR1.

4.3.4 The ccw Instruction

Each channel command word is set by a CCW instruction whose general form is

```
label  CCW   operand
```

The operand consists of four required subfields separated by commas, with no intervening blanks:

1. The first subfield consists of an absolute expression that specifies the command code. (We shall see what these codes are later on.) This expression is evaluated and placed in the first byte of the CCW.
2. The second subfield is an expression whose value indicates the data address (the location from or to which the data are to be transmitted). This value is placed in bytes 2–4 of the CCW.
3. This field contains an absolute expression whose value is used to set the flags in bits 32–36 and to place 0's in bits 37–39 of the CCW.
4. The final subfield is an absolute expression that indicates the count (the number of bytes to be transmitted). This value is placed in the last two bytes (7 and 8) of the CCW. To illustrate, we shall examine the statement

```
SEND  CCW   2,PLACE,X'68',120
```

This will set up the following CCW starting at a location named SEND:

Bit Position		*Contents*
SEND	0 – 7	00000010
SEND+1	8–31	Address corresponds to PLACE
SEND+4	32–36	01101 $_{(01101000)_2 = (68)_{16}}$
	37–39	000
SEND+5	40–47	00000000 (automatically set)
SEND+6	48–63	0000000001111000

PROBLEMS

1. Describe briefly the functions of an assembler program.
2. Distinguish between a machine instruction and an assembler instruction.
3. What is a location counter? How is it used?
4. Define an absolute term.
5. What is the difference between a literal and a self-defining term?
6. Select the valid operand expressions (assume the extended Assembler Language): W, X, Y, and Z are relocatable terms.

(a) X+2*Y	(b) *+4	(c) Y+2*−Y
(d) X+Y+Z	(e) X+Y−Z	(f) 2+X−3*Y
(g) (2+X)/3*Y	(h) =32−4*X*Y	
(i) 3+2*W−X)/5	(j) X'5F'	

7. Show what the internal representations would be for the following declarations:

(a)	DC	C'W"YTH"N"D'	(b)	DC	CL4'4AX'
(c)	DC	F'3129'	(d)	DC	F'14E−2'
(e)	DC	F'−204E1'	(f)	DC	X'SEES'
(g)	DC	XL6'EFFEIC'	(h)	DC	H'24S'
(i)	DC	D'81446'	(j)	DC	P'104'

8. Show the DC statements that would produce the following stored values:

(a)

M	I	N	E	'	S		M	I	N	E

(b)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	1	0	0	0	0	1	1	0

(c)

1	1	1	1	1	1	1	1	1	0	1	0	0	1	1	1

(d)

F	3	F	4	F	7	F	0	C	4

(e)

4	0	7	2	6	8	0	0	2	1	1	0

(f)

E	7	D	1	0	F	F	2	0	0	1	2

(g)

0	2	0	4	7	C	0	2	0	4	7	C	0	2	0	4	7	C

9. Define halfword boundaries.
10. What is the difference between DS and EQU?

Chapter 5

Binary Arithmetic and Related Machine Instructions

The primary emphasis in this and the next four chapters is on the machine instructions of System/360 Assembler Language; assembler instructions are used as necessary to provide a workable framework for examples and discussion of such machine instructions. The latter are introduced in small groups and in the context of small problems to solve. Thus both the machine instructions and many of the ways they are used are discussed concurrently.

Assembler Language and other languages are used within a System/360 environment that also includes some form of operating system. Since the software structure is equipped to handle a large variety of jobs in any combination, each job (where a job might be an assembly, compilation, problem execution, or a combination) must be accompanied by instructions to the operating system which define the type of job it is, and therefore which software components are needed. These instructions are submitted as part of the job. They are usually prepared on punched cards (*control cards*). Several types of operating systems are available for System/360, each with its own performance characteristics and control card requirements. Furthermore, for a given operating system, such control cards are not stand-

ard from installation to installation. For this reason, and because their contents are not properly a part of the Assembler Language or any other programming language, they are not included here and are not shown in the programmed examples.

The order in which Assembler Language machine instructions are presented is, to a large extent, arbitrary. Although they have been grouped in a certain way in these chapters, this has been done primarily for convenience, and no particular significance should be read into the associated chapter titles. Whereas the binary arithmetic statements form a natural grouping, for example, other statements included within this chapter could very well have been discussed in others.

In order to allow the concentration of attention on machine instructions, each programmed example is presented in the following framework:

```
Name (Label)   Operation   Operand
               START       4096
BEGIN          BALR        15,0
               USING       *,15
*   BEGINNING OF SPECIFIC EXAMPLE

.      .       .           .
*   END OF SPECIFIC EXAMPLE
               EOJ
*   BEGINNING OF DC AND DS STATEMENTS

.      .       .           .
*   END OF DC AND DS STATEMENTS
               END         BEGIN
```

The START assembler instruction is used to indicate the beginning of an assembly, to give a name to the program (if the name field is utilized), and to set the location counter to an initial value. In the framework above, the location counter (and consequently the address of the BALR machine instruction) is set to 4096. Where a problem program may start depends on how many locations in the lowest-address part of memory are used by the supervisory program and/or operating system. Our use of 4096 has no particular significance.

BALR is an RR instruction whose operand field contains the base 10 addresses of two general registers. When the second of these is 0, as it is here, the address of the first memory location following the BALR instruction is loaded into the register specified in the first operand. This is *not* the address of the USING statement, which is an assembler instruction, but of the next machine instruction. The purpose of using this instruction in this way is to load a general register at object time with a value which makes it suitable for subsequent use as a base register.

USING is an assembler instruction that tells the assembler (at assembly time) which general register to use as a base register and defines the value that can be assumed to have been loaded into that register at object time. In the above program framework, *,15 tells the assembler that at object time General Register 15 will have in it the address currently contained in the location counter. In our case this address is 4098 because the BALR instruction has an address of 4096 and occupies two bytes.

At the end of the executable portion of the example is the assembler instruction EOJ, which is used to terminate execution of the object program. Note that the statements which follow it are storage-allocation statements which do not produce executable machine instructions. This is an assembler instruction which causes a prescribed sequence of events resulting in a change in machine status.

END is an assembler instruction that tells the assembler (at assembly time) that there are no more source-program statements to assemble. The operand field may be used optionally, as it is here, to name the first executable instruction in the program. When the operand field is left blank, control is passed at execution time to the first position (the position with the lowest memory address) occupied by the problem program.

5.1 THE L, ST, AND A INSTRUCTIONS

The first operand in each of these RX instructions is a general register and the second refers to a fullword in storage. Recall that the effective address of the second operand is a sum of three parts: the contents of positions 8–31 of the base and index registers and the 12-bit displacement included in the instruction itself. At present we shall not use an index register, deferring its inclusion to a later point.

The Load (L) statement has the form

name L *n,operand*

The contents of the fullword in "operand" are duplicated in General Register *n*. For example, the statement

L 5,VAR1

will produce the results shown below.

	Before	*After*
General Register 5	00001000011001011000100111001001	01011110001100000000000000000011
VAR1	01011110001100000000000000000011	01011110001100000000000000000011

The Store (ST) statement is the reverse of L, causing the contents of the fullword in the register to be duplicated in the location designated by the second operand. Its general form is

name ST *n,operand*

For instance, the statement

```
ST    7,VAR3
```

will cause the following changes:

	Before	*After*
General Register 7	00000001000000010000000100001111	00000001000000010000000100001111
VAR3	01100110011001100110011001100111	00000001000000010000000100001111

The Add (A) instruction causes the contents of the second operand to be added algebraically to the register designated by the first operand.

In general, we write

```
name    A     n,operand
```

Given the *before* situation shown below, the following statement produces the *after* situation.

```
A     4,VAR2
```

	Before	*After*
General Register 4	00000000111100000010000001100111	00000000111110000110000100111000
VAR2	00000000000010000001000000110001	00000000000010000001000000110001

Although no use is made of it here, Add accomplishes one thing more. Depending on the result of the addition, Add automatically sets the two-bit condition code (in bit positions 34 and 35 of the current PSW) to one of the following values:

Condition Code	*Occurs if*
00	Sum = 0
01	Sum < 0
10	Sum > 0
11	Overflow

Overflow occurs if the result of the addition is larger than can be represented in the general register (sign plus 31 bits), and such occurrence causes a program interruption. The L and ST instructions do not affect the condition above.

EXAMPLE 1

Add A and B and store the result in C. A and B are binary words whose current values are 16 and 73, respectively; C is a memory field one word in length.

Solution Procedure

1. Load A into a general register (we will use 10).
2. Add B into the same general register.
3. Store the contents of that general register in C.

Solution Program

```
Name    Operation   Operand     Comment
        START       4096        SET LOCATION COUNTER TO 4096
BEGIN   BALR        15,0
        USING       *,15
        L           10,A        LOAD A INTO REGISTER 10
        A           10,B        ADD B TO REGISTER 10
        ST          10,C        STORE REGISTER 10 IN C
        EOJ                     END EXECUTION
        DS          0F          SET LOCATION COUNTER AT
                                FULLWORD
A       DC          F'16'
B       DC          F'73'
C       DS          F           AUTOMATICALLY RESERVES ONE
                                WORD
        END         BEGIN       END OF SOURCE PROGRAM
```

Recapitulating the items included in the Solution Program, including the effects of the framework previously introduced, the following operations occur at object time:

1. BALR, whose address is 4096, loads the constant 4098 into General Register 15, which will be used as a base register.
2. L reproduces the contents of memory location A in General Register 10.
3. A adds the contents of memory location B to the contents of General Register 10. Note that the assembler has no problem distinguishing this mnemonic code A from the symbol for the constant A.
4. ST duplicates the contents of General Register 10 in memory location C.
5. EOJ terminates execution of the program.

At the end of this program, A contains 16, B contains 73 and C and Register 10 both contain values of 89.

Note that we have used DC and DS assembler instructions to generate and store the data used in this little program. This allowed us to write a complete program without knowing how to handle input/output. We shall continue to do so while our primary concern remains focused on internal manipulations.

5.2 THE LH, STH, AND AH INSTRUCTIONS

The RX operations described in Sec. 5.1 may be duplicated with halfword operands by adding an H to the appropriate mnemonic code. Thus LH (Load Halfword) operates like Load, except that a halfword must be positioned in a fullword register. This is handled by the assembler, which expands the halfword by placing 16 bits between the sign bit and the 15 information bits. These additional bits are 0's or 1's, depending on the sign

of the value. To illustrate, the following statement will result in the situation shown below, assuming the indicated conditions:

```
LH   5,VARH1
```

	Before	*After*
Register 5	11111111111111110000000000000000	00000000000000000101010010101
VARH1	0010101001010101	0010101001010101

The AH instruction uses the same padding technique as LH. It affects the condition code in the same way as the A instruction.

The STH instruction stores the rightmost 16 bits of the designated register in the indicated halfword of storage. Note that it is up to the programmer to make sure that the sign remains unchanged or to adjust for any such change after storage has taken place.

EXAMPLE 2

Repeat the previous example, making A, B, and C each a halfword in length.

Solution Procedure

The procedure does not change, but A, B, and C must be defined differently and separate machine instructions must be used.

Solution Program

```
Name     Operation   Operand   Comment
         START       4096
BEGIN    BALR        15,0
         USING       *,15
         LH          10,A
         AH          10,B
         STH         10,C
         EOJ
         DS          0H        ALIGN TO HALFWORD BOUNDARY
A        DC          H'16'
B        DC          H'73'
C        DS          H
         END         BEGIN
```

The contents of A,B,C, and Register 10 will be as in the previous example, except that A, B, and C are now defined as halfwords.

5.3
THE LR, STM, AND S INSTRUCTIONS

Transfer of information from one register to another is performed with the LR instruction (also called Load). Its form is like that of the L instruction, except that both operands refer to general registers. Thus the

general statement

name LR *n*1,*n*2

represents a request to replace the contents of register *n*1 with those of *n*2. (*n*2, of course, remains unchanged.)

STM (for Store Multiple) is a very handy instruction for combining several transfer operations. It is of the RS type and will reproduce the contents of adjacent registers in contiguous storage locations. The general form is

name STM *n*1,*n*2,*operand*

where *n*1 defines the beginning of a sequence of registers and *n*2 specifies the end of that sequence. The contents of these registers are reproduced starting in the byte designated by *operand* and continuing until the sequence is completed. For instance, if we were to define storage for four fullwords under the names VAR1, VAR2, VAR3, and VAR4, the following sequence

```
        STM   6,9,VAR1
        .
        .
VAR1    DS    F
VAR2    DS    F
VAR3    DS    F
VAR4    DS    F
```

will reproduce the contents of general register 6, 7, 8, and 9 in VAR1, VAR2, VAR3, and VAR4, respectively.

S (for SUBTRACT) is an RX instruction which subtracts the second operand from the first and replaces the first by the difference. General format is like the A instructions.

Condition Code	*Condition*
00	Difference = 0
01	Difference < 0
10	Difference > 0
11	Overflow

Algebraic signs are taken into account automatically and the appropriate algebraic rules are observed. Thus when a very large negative number is subtracted from a large positive one, it is possible to have an overflow.

EXAMPLE 3

Compute X = A + B + C and Y = A + B − C, where all fields are binary words and where A, B, and C have the values of 65, −82, and 33, respectively.

Solution Procedure

Each of the required results can be developed independently of the other. A more efficient procedure, however, is to develop the partial result A + B just once and use it for both X's and Y's determination:

1. Load A into a general register.
2. Add B into the same register.
3. Transfer the partial result A + B into an adjacent register.
4. Add C into the first register.
5. Subtract C from the second register.
6. Store the results in X and Y.

Solution Program

```
Name    Operation   Operand     Comment
        START       4096
BEGIN   BALR        14,0        REGISTER 14 IS THE BASE
                                REGISTER
        USING       *,14
        L           10,A        A IS IN REGISTER 10
        A           10,B        A + B IS IN REGISTER 10
        LR          11,10       A + B ALSO IN REGISTER 11
        A           10,C        A + B + C IN REGISTER 10
        S           11,C        A + B - C IN REGISTER 11
        ST          10,X        A + B + C IN X
        ST          11,Y        A + B + C IN Y
        EOJ
        DS          0F
A       DC          F'65'
B       DC          F'-82'
C       DC          F'33'
X       DS          F           RESERVE A WORD FOR X
Y       DS          F           RESERVE THE NEXT WORD FOR Y
        END         BEGIN
```

By using STM, we can reduce the length of the program by one statement. Since X and Y are in contiguous locations (then DS's are adjacent) and their potential contents are in adjacent registers (10 and 11), we can replace the two ST statements with a single STM:

```
        STM     10,11,X
```

5.4 THE LM, AR AND SR INSTRUCTIONS

Multiple loading (i.e., from contiguous storage to adjacent registers) is obtained with the LM instruction:

```
name    LM      n1,n2,operand
```

As with STM, *n*1 and *n*2 represent the register numbers which begin and end the desired sequence and *operand* specifies the address of the first byte to be loaded.

AR and SR operate like the corresponding RX instructions (A and S) except that both operands are now registers. Thus when we write

name AR *n*1,*n*2

the contents of register *n*2 are added to those of *n*1. Similarly,

name SR *n*1,*n*2

causes the value in *n*1 to be reduced by that in *n*2. Condition codes are affected as with the A and S instructions.

EXAMPLE 4

Compute X = A − B and Y = B + C, where all fields are binary words and where A, B, and C have the values −105, 89, and −150, respectively.

Solution Procedure

If both computations were carried out independently, six instructions would be required: L, S, ST, L, A, and ST:

```
Name    Operation   Operand     Comment
        START       4096
BEGIN   BALR        13,0
        USING       *,13
        L           10,A        −105 IN REGISTER 10
        S           10,B        (−105−89) IN REGISTER 10
        ST          10,X        (−105−89) IN X
        L           10,B        89 IN REGISTER 10
        A           10,C        89+(−150) IN REGISTER 10
        ST          10,Y        89+(−150) IN Y
        EOJ
        DS          0F
A       DC          F'−105'
B       DC          F'89'
C       DC          F'−150'
X       DS          F
Y       DS          F
        END         BEGIN
```

By making use of STM, LM, SR, and AR, together with the fact that X and Y occupy contiguous storage locations, we can eliminate two instructions:

Solution Program

```
Name    Operation   Operand     Comment
        START       4096
BEGIN   BALR        13,0
        USING       *,13
```

```
Name    Operation   Operand    Comment
        LM          10,12,A    -105 IN REG. 10, 89 IN REG. 11,
                               -150 IN REG. 12
        SR          10,11      -105-89 IN REG. 10
        AR          11,12      89+(-150) IN REG. 11
        STM         10,11,X    (-105-89) IN X, 89+(-150) IN Y
        EOJ
```

5.5
THE LPR, LNR, AND LCR INSTRUCTIONS

These three RR instructions allow the programmer to introduce certain changes in the contents of a register and place those changed contents in another register with one statement.

The Load Positive Register (LPR) instruction, having the general format

name LPR *n*1,*n*2

causes the contents of *n*2 to be duplicated in *n*1 with a positive sign, irrespective of *n*2's sign. *n*2, of course, remains unchanged. If *n*2 is negative, complementation occurs automatically as part of execution. The condition code is affected as follows:

Condition Code	*Occurs if*
00	*n*1 is 0 after execution of LPR.
01	——
10	Execution is routine.
11	Overflow (*n*2 is maximum negative value).

The LNR (Load Negative Register) instruction always places a negative number in the register specified by the first operand. If the number in the second register is positive, it will be automatically complemented. Using the generalized statement

name LNR *n*1,*n*2

for reference, the condition codes are set as follows:

Condition Code	*Occurs if*
00	*n*1 is 0 after execution of LNR.
01	Execution is routine.
10	——
11	——(Overflow is impossible).

The LCR (Load Complement Register) instruction changes the sign and places the complement of the second named register (*n*2) in the first (*n*1). If, in general,

name LCR *n*1,*n*2

then the condition codes will be set as follows:

Condition Code	*Occurs if*
00	*n*1 is 0 after execution of LCR.
01	*n*1 is negative after execution.
10	*n*1 is positive after execution.
11	overflow (*n*2 is maximum negative value).

If we use the simple trick of specifying the same register in both operands, then LPR becomes, in effect, a "set sign positive" instruction. LNR acts like a "set sign negative" instruction and LCR behaves like a "change sign" instruction.

EXAMPLE 5

A and B are binary words whose values are 85 and 49 respectively. Compute X = |A| − |B| and Y = −X.

Solution Procedure

1. Load A and B into consecutive general registers.
2. Convert A into its absolute value (set its sign positive).
3. Convert B into the negative of its absolute value (set its sign negative).
4. Add |A| and − |B| to produce X.
5. Reverse the sign of X to produce Y.
6. Store the results in X and Y.

Solution Program

```
Name    Operation   Operand     Comment
        START       4096
BEGIN   BALR        14,0
        USING       *,14
        LM          10,11,A     85 IN REG. 10, -49 in 11
        LPR         10,10       85 IN REG. 10 (|A|)
        LNR         11,11       -49 IN REG. 11 (-|B|)
        AR          10,11       |A|+-|B| IN REG. 10
        LCR         11,10       -(|A|+-|B|) IN REG. 11
        STM         10,11,X
        EOJ
        DS          0F
A       DC          F'85'
B       DC          F'49'
```

```
Name    Operation   Operand   Comment
X       DS          F
Y       DS          F
        END         BEGIN
```

5.6 THE MR, M, AND MH INSTRUCTIONS

Fixed-point multiplication can be performed with one of three instructions, depending on prevailing circumstances.

The MR (Multiply Registers) instruction assumes that all of the transactions will take place in registers and will involve fullwords. Multiplication proceeds according to the following prescribed procedure. The product is developed in an adjacent pair of registers, the rightmost of which must be odd-numbered. In order for this to happen, the programmer must arrange for the multiplicand to be in the odd-numbered register. The multiplier may be in any other register. As multiplication proceeds, the multiplicand is destroyed. Depending on the sizes of the two factors, the product may or may not carry into the high-order (even-numbered) member of the pair of registers. Thus 63 bits are available for the product value (the remaining one, of course, contains the sign). Since each factor starts out no larger than 31 bits (plus a sign), the largest possible product value is a 62-bit number, so that there is no possibility of overflow in the product field. Hence the MR instruction does not affect the condition code.

The general form for this instruction is

name MR *mcand,mplr*

where *mcand* represents the leftmost member of the register pair and therefore must be an *even* number. The other register, the odd-numbered one which actually contains the multiplicand, is not directly mentioned here. The second operand, *mplr*, specifies the register containing the multiplier. In other words, the contents of register (*mcand* + 1) are multiplied by those of *mplr* and the resulting product appears in (*mcand* + 1), with the high-order bits carrying over into *mcand* if the value is sufficiently large. For this reason it is usually a good idea to fill *mcand* with zeros before multiplication. Incidentally, register 0 is considered even-numbered.

To illustrate the use of MR, we shall embed it in a *typical* sequence of statements which prepare the registers and store the result. Our assumptions are as follows:

1. Our multiplicand is 10,000 and is stored in X.
2. Our multiplier is 5,000 and is stored in Y.
3. Registers 4 and 5 will be used for the product.
4. Register 9 will be used for the multiplier.

5. We don't know how large the product is going to be, so we reserve a doubleword (call it Z) for it.

Our sequence would appear as follows:

```
Name    Operation   Operand     Comment

                .
                .
        LM      4,5,ZEROS   REGISTER 4 IS SET TO ZERO
                            AND REGISTER 5 HAS X
        L       9,Y         REGISTER 9 HAS Y
        MR      4,9         X*Y IS IN REGISTERS
                            4 AND 5
        STM     4,5,Z       PRODUCT X*Y IS IN Z

                .
        DS      0F          ALIGN ON FULLWORD BOUNDARY
ZEROS   DC      F'0'
X       DC      F'10000'
Y       DC      F'5000'
        DS      0D          ALIGN ON DOUBLEWORD BOUNDARY
Z       DS      D
```

Before Multiplication:

Registers 4-5:

00000000000000000000000000000000	00000000000000000001001110001000

Register 9

00000000000000000001001110001000

After Multiplication:

Registers 4-5:

00000000000000000000000000000000	00000010111110101111000010000000

If we assumed advance knowledge of the fact that the product would not exceed a fullword, we would have reserved only one word for it and would have written ST 5,Z instead of STM 4,5,Z.

Although no overflow is possible, an interrupt can still occur as a result of an MR. This may happen when the first operand is not an even-numbered register.

The M (Multiply) instruction works the same way as MR except that the second operand is now a storage location. Use of this instruction would change our previous sequence to

```
        LM      4,5,ZEROS
        M       4,Y
        STM     4,5,Z
```

and so on. As is the case with MR, no condition codes are set.

If the program calls for a multiplication in which both multiplicand and multiplier are halfwords, only one register need be involved, since the largest possible product in this case can be accommodated in a fullword with no overflow possible. The MH (Multiply Halfword) instruction does this by using a multiplicand in a single register and developing the product in that same register, thereby destroying the multiplicand. The following sequence exemplifies this.

```
        L       6,ZEROS
        LH      6,X
        MH      6,Y
        ST      6,Z
        .
        .
        DS      0F
ZEROS   DC      F'0'
Z       DS      F
X       DC      H'156'
Y       DC      H'12'
```

Before Multiplication (after LH):

Register 6: 00000000000000000000000010011100 Y: 0000000000001100

After Multiplication:

Register 6: 00000000000000000000011101010000 Z: 0000011101010000

EXAMPLE 6

A and B are binary words whose values are 25 and 31, respectively; C is a binary halfword whose value is 16. Compute $X=A^2*B*C$ and store it as a binary word.

Solution Program

```
Name    Operation   Operand     Comment
        START       4096
BEGIN   BALR        14,0
        USING       *,14
        L           11,A        25 IN REGISTER 11
        MR          10,11       25*25 DEVELOPED IN 10 AND 11
        M           10,B        25*25*31 DEVELOPED IN 10 AND 11
        MH          10,C        25*25*31*16 DEVELOPED IN 10
                                AND 11*
        ST          11,X        FINAL RESULT IN X
        EOJ
        DS          0F          ALIGN TO FULLWORD BOUNDARY
A       DC          F'25'
B       DC          F'31'
C       DC          H'16'       NO ALIGNMENT NECESSARY
        DS          0F          REALIGN TO FULLWORD BOUNDARY
X       DS          F
        END         BEGIN
```

The halfword would automatically be expanded with high-order zeros so that the multiplication could proceed.

5.7 THE SLDA, SRDA, D, AND DR INSTRUCTIONS

Certain types of multiplications can be performed more rapidly by taking advantage of System/360's internal storage methods. Since we are dealing with binary words here, we can perform multiplications and/or divisions by 2 simply by shifting the word with respect to some fixed binary point (which in this case is at the right-hand end of the word). To understand this fully, let us suppose for a moment that we are dealing with an 8-bit binary word containing the equivalent of 62:

00111110

and we had a way of shifting it one bit to the left, losing the leftmost bit and adding a 0 at the right: the resulting value would be

01111100

or 124. Shifting left once again would give us

11111000

or 248. Had we gone in the other direction we would have been dividing by two each time we shifted one place to the right.

The SLDA (Shift Left Double) instruction allows us to perform such multiplications by two on the contents of an even/odd pair of registers, treating them as a doubleword. Its general form is

name SLDA *n1,m*

where n1 specifies the even-numbered member of the register pair and the last six bits of *m* specify the number of places to be shifted to the left. Thus,

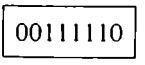
SLDA 8,6

will cause the contents of registers 8 and 9 to be shifted 6 places to the left (in effect, multiplying that doubleword by 2^6 or 64). Note that although SLDA is an RS instruction, the assembler takes care of the missing third operand.

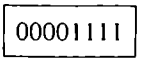
The sign bit in the leftmost (even-numbered) register is not shifted, but the first bit in the odd-numbered register is treated as a data bit and not a sign bit, so that it participates in the shift. SLDA affects the condition code as follows:

Condition Code	*Occurs if*
00	Result after shift is 0.
01	Result after shift is < 0.
10	Result after shift is > 0.
11	Overflow (if the bit shifted out of the leftmost data bit differs from the sign bit).

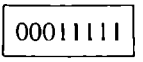
SRDA (for Shift Right Double) shifts the contents of an even/odd register pair to the right. The sign bit in the odd-numbered register is treated like a data bit, i.e., the two registers are considered as a 63-bit + 1 sign bit field. The high order end is filled in with sign bits and a corresponding number of bits are lost at the low order end. This causes an inexact result whenever the contents represent an odd integer because fractional quantities cannot be handled. Wherever this is significant with respect to the problem being solved, loss of fractional remainders can be avoided by the use of scale factors or by turning to floating-point arithmetic. Thus if we assume an eight-bit field for illustrative purposes, a one-bit right shift of



produces a value of



which is .5 (or 2^{-1}) times the previous one. The same operation on



also yields

00001111

which is not exactly half.

SRDA cannot cause an overflow condition, but has the following effect on the condition code:

Condition Code	*Occurs if*
00	Result = 0.
01	Result < 0.
10	Result > 0.

D (for Divide) is an RX instruction that operates by a process of repeated subtraction. That is, the quotient is the greatest integral number of times that the divisor can be subtracted from the dividend, and the remainder is what is left over after the subtractions are carried out. The first operand specifies the even-numbered member of the even/odd register pair containing the dividend, and the second operand gives the location of the divisor in memory. The remainder, which has the same sign as the dividend, is contained in the even-numbered register, and the quotient, whose sign is determined by the rules of algebra, is developed in the odd-numbered register. If the quotient or the remainder is zero, however, its sign is always positive.

The Divide instruction is often used in conjunction with a Shift instruction, which is included to adjust the dividend so that the quotient and remainder are generated with the desired number of places. A division operation does not set the condition code. The following sequence illustrates the use of this instruction:

```
         LM     2,3,VIDIV
         D      2,VDS

         .
         DS     0F
VIDIV    DC     F'0'
         DC     F'1133'
VDS      DC     F'44'
```

Before Division:

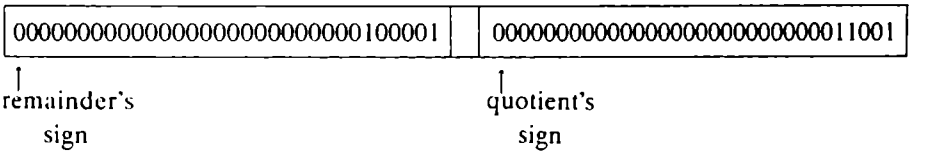
Registers 2-3

00000000000000000000000000000000	00000000000000000000010001101101

↑
sign

After Division:

Registers 2-3



EXAMPLE 7

A and B are binary words whose values are 53 and 51, respectively. Compute 128A/B, storing the quotient in Q and the remainder in R.

Solution Procedure

1. Load A into an odd-numbered general register.
2. Set to zero the even-numbered general register whose address is one less. (This procedure applies to positive dividends only; the general procedure is discussed later.)
3. Shift the contents of the even/odd register pair seven bit-positions to the left.
4. Divide by B.
5. Store the quotient in Q and the remainder in R.

Solution Program

```
Name    Operation   Operand     Comment
        START       5000        THIS TIME WE STARTED AT 5000.
                                PERFECTLY FINE.
GO      BALR        15,0
        USING       *,15
        L           11,A        53 IN REG. 11
        SR          10,10       ZERO IN REGISTER 10
        SLDA        10,7        53*128 IN REGISTERS 10-11
        D           10,B        QUOTIENT IN REG. 11, REMAINDER
                                IN REG. 10
        STM         10,11,R
        EOJ
        DS          0F
A       DC          F'53'
B       DC          F'51'
R       DS          F
Q       DS          F
        END         GO
```

General Register 10 is set to zero by subtracting it from itself. This will work in this example or in any division in which the dividend is known to be positive. If the dividend is negative, however, the even-numbered register should be set to ones. If the sign is not known in advance—that is, if it is dependent on the data—the dividend is tested and the register set according to the result. Instructions for conducting such a test are introduced later.

DR (for Divide) operates exactly like D except that the divisor specified by the second operand is in a general register rather than in memory.

EXAMPLE 8

A and B are binary words whose values are 158 and 23, respectively. Compute .5A/B and store the remainder in X and the quotient in Y.

Solution Procedure

1. Load A and B into general registers such that A is in an odd-numbered register.
2. Set to zero the even-numbered register whose address is one less than that of A.
3. Shift the even/odd register pair one position to the right to multiply A by .5.
4. Divide .5A by B.
5. Store the remainder and quotient in X and Y.

Solution Program

```
*Name     Operation   Operand     Comment
          START       4096
ORIGIN    BALR        15,0
          USING       *,15
          LM          11,12,A     158 IN REG. 11, 23 IN REG. 12
          SR          10,10       ZERO IN REG. 10
          SRDA        10,1        158/2 IN REG. 11, ZERO IN 10.
          DR          10,12       REMAINDER IN REG. 10,
                                  QUOTIENT IN 11.
          STM         10,11,X
          EOJ
          DS          0F
A         DC          F'158'
B         DC          F'23'
X         DS          F
Y         DS          F
          END         ORIGIN
```

5.8 THE SLA, SRA, AND SH INSTRUCTIONS

When it is appropriate to multiply or divide the contents of a single register by a power of 2, the SLA and SRA instructions can be used in exactly the same way as described for SLDA and SRDA. (Of course, in this case the first operand specifies the actual single register in which the shift is to occur.) The condition codes are set in exactly the same way as with the corresponding double shift instructions.

SH (Subtract Halfword) is an RX instruction that bears the same relation

to S as AH does to H. When subtraction is actually executed, the second operand will be complemented automatically. Condition codes are set as follows:

Condition Code	*Occurs if*
00	Difference = 0.
01	Difference < 0.
10	Difference > 0.
11	Overflow.

EXAMPLE 9

A is a binary word whose value is 75, and B is a binary halfword whose value is −59. Compute .25(32A − B) and store the result in C.

Solution Procedure

1. Load A into a general register.
2. Shift left five positions.
3. Subtract B.
4. Shift right two positions.
5. Store the result in C.

Solution Program

```
*Name     Operation   Operand     Comment
          START       8000        OK. SO THIS TIME IT'S 8000.
BEGIN     BALR        15,0
          USING       *,15
          L           10,A        75 IN REGISTER 10
          SLA         10,5        32*75 IN REGISTER 10
          SH          10,B        32*75-(-59) IN REGISTER 10
          SRA         10,2        .25(32*75 - (-59)) IN REGISTER 10
          ST          10,C
          EOJ
          DS          0F
A         DC          F'75'
C         DS          F
B         DC          H'-59'
          END         BEGIN
```

PROBLEMS

A, B, C, and D are binary words whose values are 33, −53, 45, and 72, respectively. Write Assembler Language programs to carry out the following computations and to store the results in memory.

1. X = A + B + C + D
2. X = (A − 2B + 2C − D)/4
3. X = A + B; Y = B + C; Z = C + D

4. $X = 37A - 52B$
5. $X = |39C - 35D|$
6. $X = (B + C)^4$
7. $X = A + 4B + 8C + 16D$
8. $X = (A + B)^2/(C + D)$
9. $X = (A^2 + B^2)/(C^2 + D^2)$
10. $X = (A^2 - AB + B^2 - A - B)/(C + 2D)$

Chapter 6

Branching

This chapter is concerned with those instructions that provide branching capabilities—that is, allow decision making and the pursuit of alternate lines of processing. Even here the choice of which instructions belong in this category is somewhat arbitrary. Add, for example, sets the condition code and this setting may be used in subsequent decision-making instructions.

Some of the instructions discussed here are closely related to those already illustrated in the previous chapter. Others, however, are more directly tied to decimal, hexadecimal, and logical operations not yet encountered, while still others have no special orientation. Because each instruction is used and described in context, no particular difficulty should be experienced. It might be well to review both the data formats and the instruction formats given in Chapters 2 and 3, should any such difficulty arise. The primary though not necessarily exclusive purposes of the instructions introduced in this chapter are to set the condition code, operate according to a previous setting of the condition code, or operate according to other variable conditions.

6.1 THE LTR AND BC INSTRUCTIONS

System/360 instructions include a variety of operations which combine the setup and performance of a test in a single step. Once a test has been conducted in this manner, the programmer can dictate the choice of alternative operation sequences based on its outcome.

The LTR (Load and Test Register) is an RR instruction of this type. Execution of this instruction, whose general form is

name LTR *n1,n2*

causes the contents of register *n2* to be reproduced in register *n1* and the condition code to be set according to whether the contents are less than, equal to, or greater than zero:

Condition Code	*Produced When*
00	Contents = 0.
01	Contents < 0.
10	Contents > 0.
11	Not used.

There is no need to involve two registers in this operation unless it serves some other useful purpose. For example, it is legitimate to write

```
LTR     12,12
```

whereupon the contents of register 12 will remain as they are, but the condition code will be set as indicated previously.

Once the condition code has been set, we can specify a sequence change based on its value by means of the BC (Branch on Condition) instruction. The general form is

name BC *m,operand*

where *operand* represents the destination of the branch and is usually a statement label, and *m* is a decimal integer constant (unsigned) which specifies the condition under which the branch is to occur. At first glance it would appear that *m* would be either 0, 1, 2, or 3, corresponding to the four possible condition code values. However, it is much more flexible than that. Although the programmer specifies a decimal integer, the system translates that into a binary number which is compared to a special table. This allows the specification of branches on combinations of the basic conditions as follows. Recall that the condition code, regardless of the mechanism by which it is set, is always contained in bits 34–35 of the current PSW. When the condition code is tested (as it is by a BC instruction) it is not a direct test of these two bits. Instead, their value is juxtaposed against a fixed table of four-bit entries.

When Condition Code Is	*Corresponding Table Value*
00	1000
01	0100
10	0010
11	0001

It is the *table value* (its decimal equivalent) and *not* the actual condition code that is specified by the first operand in the BC statement. Thus when we want to branch to a statement labeled NOMORE on a condition code of 00, we would write

```
BC      8,NOMORE
```

The reason for this indirect type of comparison is that additional mechanisms are provided for branching on combinations of conditions. These are actually fairly simple. By using 1, 2, 4, and 8 to represent the condition codes, any number between 1 and 15 can then be obtained as a unique sum of these, and therefore represents a particular combination of condition codes. For instance, an operand of 3 (which can only be 2 + 1 in this context) would indicate a branch if the condition code is 10 (table value of 2) or 11 (table value of 1). Similarly, the statement

```
BC      11,THERE
```

will cause a branch to the statement labeled THERE if the table value is either 8, 2, or 1 (since 11 = 8 + 2 + 1). Thus this branch will occur if the condition code is anything but 01. Should the condition code be 01 (table value of 4) the branch will be ignored and execution will continue in normal sequence. Table 6.1 summarizes the possible BC first operand values and the conditions they represent. It is seen that a BC with a 0, *operand* will never branch while a BC 15, *operand* will always branch.

The BCR instruction is the RR counterpart of BC, with the difference being in the second operand. For the BCR instruction the second operand specifies

TABLE 6.1 BC FIRST OPERANDS

First Operand	Condition Code	Correlation
0	Nonexistent	—
1	11	Overflow
2	10	Positive
3	10 or 11	Positive or overflow
4	01	Negative
5	01 or 11	Negative or overflow
6	01 or 10	Nonzero
7	01 or 10 or 11	Nonzero or overflow
8	00	Zero
9	00 or 11	Zero or overflow
10	00 or 10	Not negative
11	00 or 10 or 11	Not negative or overflow
12	00 or 01	Not positive
13	00 or 01 or 11	Not positive or overflow
14	00 or 01 or 10	Anything but overflow
15	00 or 01 or 10 or 11	Unconditional

the register containing the address of the branch destination. For instance,

```
BCR     11,6
```

specifies that a branch is to occur to the address given in register 6 if the condition code is either 00, 10, or 11 (see Table 6.1).

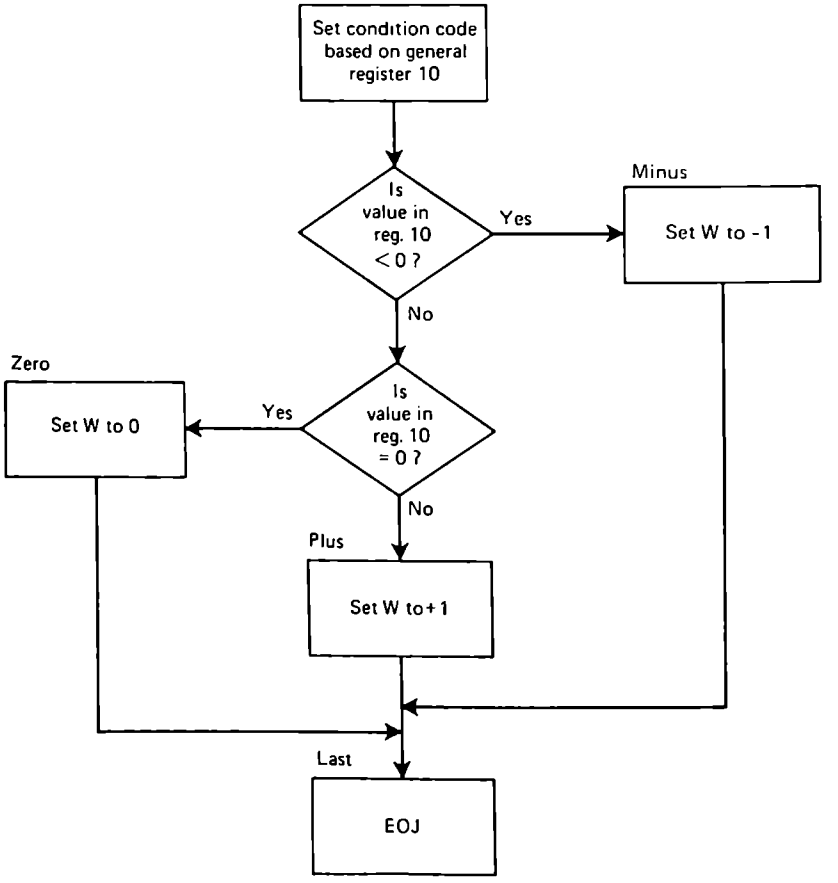
EXAMPLE 10

Assume that prior processing has left a binary word in General Register 10. W is a binary word in memory. Set W equal to -1, 0, or $+1$ according to whether the word in General Register 10 is negative, zero, or positive, respectively.

Solution Procedure

1. Set the condition code based on whether the word in General Register 10 is negative, zero, or positive.
2. Test the condition code and branch to one of three different instructions.
3. Set W to -1, 0, or $+1$ and branch, if necessary, to the EOJ statement.

Figure 6.1



Solution Program

```
Name    Operation   Operand     Comment
        START       10000
BEGIN   BALR        15,0
        USING       *,15
        LTR         10,10       SET CONDITION CODE
        BC          4,MINUS     BRANCH ON NEGATIVE VALUE
        BC          8,ZERO      BRANCH ON ZERO VALUE
PLUS    L           11,ONE      1 IN REGISTER 11
        ST          11,W        1 IN W
        BC          15,LAST     UNCONDITIONAL BRANCH TO
                                PROGRAM END
MINUS   L           11,ONE      1 IN REGISTER 11
        LCR         11,11       -1 IN REGISTER 11
        ST          11,W        -1 IN W
        BC          15,LAST
ZERO    SR          11,11       0 IN REGISTER 11
        ST          11,W        0 IN W
LAST    EOJ
        DS          0F
ONE     DC          F'1'
W       DS          F
        END         BEGIN
```

The flowchart for the program is shown in Figure 6.1.

6.2 THE TM INSTRUCTION

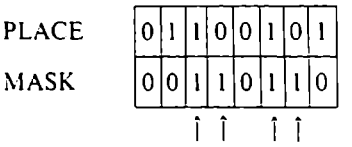
This SI instruction (Test under Mask) allows the programmer to test a byte in memory (designated by the first operand) bit by bit against an 8-bit immediate second operand called a mask. Depending on the composition of the mask, certain bits of the byte in memory are tested according to the following selection criterion. If a particular mask bit is 1, the corresponding data bit is tested; if the mask bit is zero, the corresponding data bit is ignored. Three basic types of outcomes are reported:

(a) If all the tested bits are all 0's, the condition code is set to 00 (table value of 8).
(b) If all the tested bits are all 1's, the condition code is set to 11 (table value of 1).
(c) If the tested bits are mixed, the condition code is set to 01 (table value of 4).

Let us see how this works in a simple illustrative case: Suppose we said

```
TM      PLACE,X'36'
```

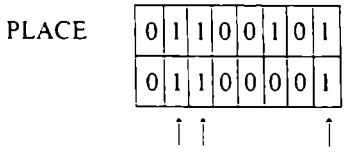
(hexadecimal form is generally used for convenience) and PLACE contained 01100101. Since x'36' is equivalent to 00110110 our test would be as follows:



Since the mask has 1's in bits 2,3,5, and 6 (counting from bit 0) it is only those bits that will be tested in the data. Thus, since bits 2 and 5 are 1's and 3 and 6 are 0's, the results are mixed, causing the condition code to be set to 01. Had we written

```
TM    PLACE,X'61'
```

our test would be



and the resulting condition code would be 11.

EXAMPLE 11

Z is the address of a byte in memory. Test its odd-numbered bits (0 is considered even). If all are 0, set the fullword X equal to 6, if all are 1's set X to 6 and fullword Y to 12. Otherwise set X and Y to 0.

Solution Procedure

1. Match the byte at Z against a mask.
2. Use the resultant condition code to define the appropriate branches.

Solution Program

```
Name    Operation  Operand   Comment
        START      4096
BEGIN   BALR       15,0
        USING      *,15
        TM         Z,X'55'   TEST ODD NUMBERED BITS
        BC         1,SETY    BRANCH IF TEST BITS ARE 1's
        BC         8,SETX    BRANCH IF TEST BITS ARE 0's
        L          10,X      X IN REGISTER 10
        SR         10,10     0 IN REGISTER 10
        ST         10,X      0 IN X
        ST         10,Y      0 IN Y
        BC         15,LAST   BRANCH TO PROGRAM END
```

```
Name    Operation   Operand     Comment
SETY    L           11,C6       6 IN REGISTER 11
        SLA         11,1        12 IN REGISTER 11
        ST          11,Y        12 IN Y
        SRA         11,1        6 IN REGISTER 11
SETX    ST          11,X        6 IN X
LAST    EOJ
        DS          0F
C6      DC          F'6'
X       DS          F
Y       DS          F
Z       DS          C
        END         BEGIN
```

The flowchart is shown in Figure 6.2.

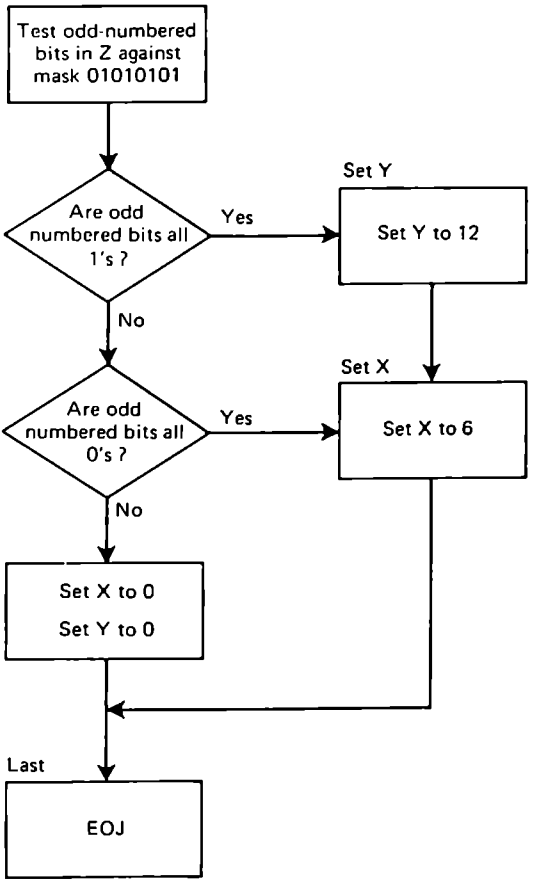


Figure 6.2

6.3
THE C, CR, AND CH INSTRUCTIONS

Instead of comparing the contents of a storage location or register to a reference of zero, the programmer, if he wishes, may test the relative values of two operands. This is done by means of one of the Compare instructions.

Comparison between a word in a register and one in storage is performed with the RX type C (Compare) instruction, whose general form is

name C *n1,operand*

with *n1* and *operand* having their previous meanings. Based on the comparison, the condition code is set according to the following schedule:

Outcome	*Condition Code*	*Table Value*
n1 = *operand*	00	8
n1 < *operand*	01	4
n1 > *operand*	10	2

For example, if register 6 contains a value of 740 and a location named B contains a value of 389, the sequence

```
C   6,2*B
BC  4,BIGB
```

will cause a condition code of 01 (a table value of 4) to be set, thus allowing the branch to be executed, and the next instruction to be processed would be at location BIGB.

The same type of comparison, but with both operands in registers, can be specified with the CR (Compare Register) instruction. Condition codes are set as in the C instruction, with a second register replacing the operand.

When a register-storage comparison is to be made with the latter operand being a halfword, the CH (Compare halfword) instruction is used. As part of its execution, the halfword operand is expanded to a fullword by filling in the extra high-order bits with the value in the sign bit. Condition codes and corresponding table values are then set as before.

EXAMPLE 12

A, B, X, and W are binary words, and Z is a binary halfword. Set W equal to 0 if X − B is greater than W + 1 and greater than Z and if X is not less than A. Otherwise, set W equal to 1.

Solution Procedure

1. Form X − B, W + 1, and X in general registers.
2. If X − B is not greater than W + 1, set W = 1; otherwise, go on to the next comparison.

3. If X − B is not greater than Z, set W = 1; otherwise, go on to the next comparison.
4. If X is less than A, set W = 1; otherwise, set W = 0.

Solution Program

```
Name      Operation   Operand     Comments
          START       5000
BEGIN     BALR        14,0
          USING       *,14
          LM          10,11,W     W IN REGISTER 10, X IN 11
          LR          12,11       X IN REGISTER 12
          S           11,B        X-B IN REGISTER 11
          A           10,ONE      W+1 IN REGISTER 10
          CR          11,10       COMPARE X-B TO W+1
          BC          12,SET1     BRANCH TO SET1 IF X-B<W+1
          CH          11,Z        X-B>W+1. COMPARE X-B TO Z
          BC          12,SET1     BRANCH TO SET1 IF X-B<Z
          C           12,A        X-B>W+1 AND X-B>Z
          BC          4,SET1      BRANCH TO SET1 IF X<A
SET0      SR          11,11       X IS NOT LESS THAN A
                                  SET REG. 11 to 0
          ST          11,W
          BC          15,OVER
SET1      L           11,ONE  SET X TO 1
          ST          11,W    SET W TO 1
OVER      EOJ
          DS          0F
A         DS          F
B         DS          F
W         DS          F
X         DS          F
ONE       DC          F'1'
Z         DS          H
          END         BEGIN
```

The flowchart for this program is shown in Figure 6.3.

6.4 ASSEMBLER LANGUAGE SUBROUTINES

System/360 Assembler Language provides some easily used but very effective mechanisms for setting up branches to and from subroutines. Once a particular sequence of operations has been identified as being useful in several places in a program and the corresponding series of instructions has been written, they can be embedded somewhere in the program as an entity. Since it is possible to branch to that sequence from any portion of

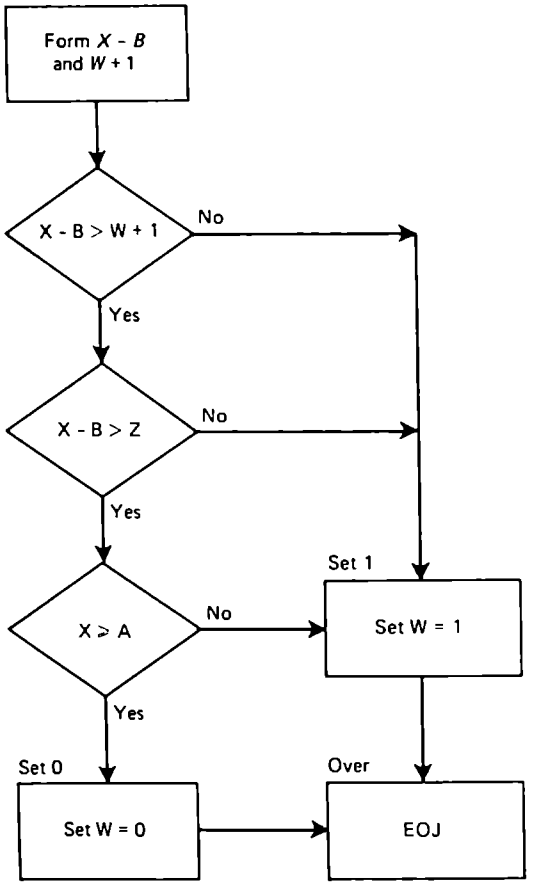


Figure 6.3

the program with the assurance that the normal sequence will be resumed after subroutine execution, the subroutine need appear only once.

6.4.1 Basic Sequence-Changing Concepts

We shall work our way into subroutines and their linkages with the rest of the program by first looking at an example containing a simple branch to a particular sequence of instructions followed by a return branch. This is not a subroutine and the program could have been written just as easily as an unbroken sequence. However, it will be useful for us to establish the principle.

EXAMPLE 13

A, B, C, X, and Y are binary words. Write a program that computes Y = A + B, branches to a location called CALC to compute X = Y + C, and then returns to the

first sequence of instructions to compute the arithmetic mean of X and Y and store it in A.

Solution Procedure

1. Load A; add B; store in Y.
2. Branch to CALC.
3. Load Y; add C; store in X.
4. Branch back to the first sequence.
5. Load X; add Y; shift right 1; store in A.

Solution Program

```
Name     Operation  Operand     Comment
         START      4096
BEGIN    BALR       15,0
         USING      *,15
         L          10,A        A IN REGISTER 10
         A          10,B        A+B IN REGISTER 10
         ST         10,Y        A+B IN Y
         BC         15,CALC     UNCONDITIONAL BRANCH TO CALC
RETURN   L          10,Y
         A          10,X        Y+X IN REGISTER 10
         SRA        10,1        (Y+X)/2 IN REGISTER 10
         ST         10,A
         EOJ
CALC     L          10,Y
         A          10,C
         ST         10,X        Y+C IN X
         BC         15,RETURN   UNCONDITIONAL BRANCH
                                TO RETURN
         DS         0F
A        DS         F
B        DS         F
C        DS         F
X        DS         F
Y        DS         F
         END        BEGIN
```

The flowchart is shown in Figure 6.4.

The branch to CALC and the branch back to RETURN disrupt the natural sequence of instructions and place CALC and the three instructions that follow it in an unusual position. To branch out and back is straightforward, only unconditional branch instructions being needed, but to do so just once brings no advantage. Consider the possibility, however, that some algorithms require the repetitive execution of a special set of instructions. The usual method of programming is to write this set of instructions only once as a subroutine and to branch to and from it whenever the particular function it serves is required.

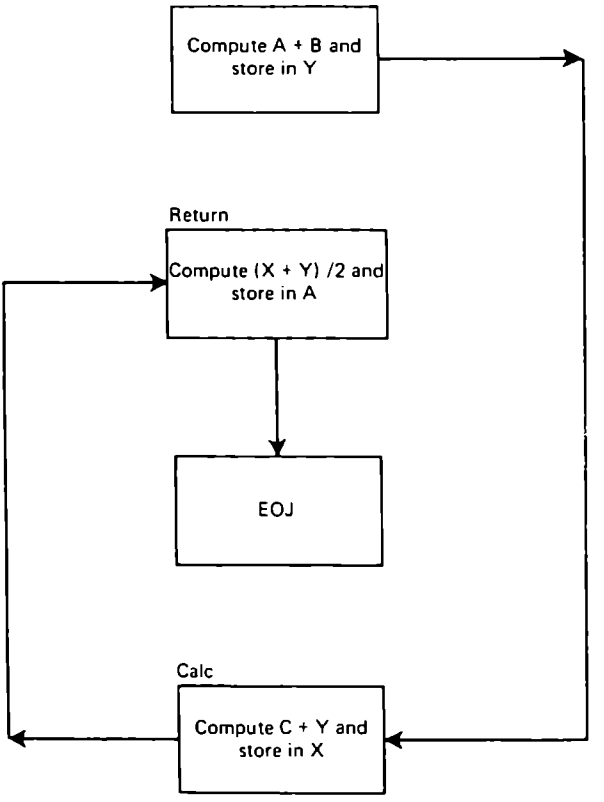


Figure 6.4

6.4.2 Subroutine Linkages

Branching to a subroutine from numerous locations in a program is direct; after all, its first instruction is always in the same location, and an unconditional branch is all that is needed. Returning to the main-line program is quite another matter, however, because there are as many points of return as there are entries to the subroutine. To carry this out, an unconditional branch is no longer sufficient. One way of linking a subroutine to a main program, illustrated in the next example, is to branch to a subroutine with information that the subroutine can use to negotiate the return to the proper point.

Facilities to provide such information are available to the programmer by means of the BAL (Branch and Link) instruction. This RX instruction has the general form

name BAL *n1,operand*

where *n1* specifies a particular register and *operand* is the address of the first instruction of the subroutine to which it is desired to branch. When

the instruction is executed, the last 32 bits of the current PSW are placed in *n*1 and the program branches unconditionally to the address given in the second operand, with execution proceeding from that point. Since the contents of the last 32 bits of the current PSW include the address of the next instruction in normal sequence (the one that would have been executed had there been no branch), the subroutine can determine where it is to return after it executes.

BALR (also standing for Branch and Link) is the RR counterpart of the BAL instruction. The basic difference is that the branch destination is stored in the register specified by the second operand. Thus, for example, the statement

```
BALR  6,5
```

says, in essence, "store the last 32 bits of the current PSW in register 6 and branch to the address given in register 5." If the second operand is zero, no branch occurs. We have been using BALR in this latter form to load a base register without causing a branch to take place.

EXAMPLE 14

Write a program that computes W = 2A + 2B + 2C by using a subroutine whose function is to double whatever number it is given and by branching to this subroutine three times to double A, B, and C in turn.

Solution Procedure

1. Load A into a general register and branch to the doubling subroutine.
2. Store 2A in W.
3. Load B into a general register and branch to the doubling subroutine.
4. Add 2A and 2B and store the sum in W.
5. Load C in a general register and branch to the doubling subroutine.
6. Add 2C to 2A + 2B and store the sum in W.

By loading the quantity to be doubled into a specific register, the subroutine can be written to process it as desired. By leaving the result in a specific register (in this example the same register is used for both purposes), the main program can be written to make use of it. BAL is used to branch to the subroutine and simultaneously place a return address in an accessible location, and BCR is used to branch back to this saved address. Note that in the program, as shown here, we are assuming that A, B, and C receive values somehow.

Solution Program

```
Name      Operation   Operand     Comment
          START       8192
BEGIN     BALR        15,0
          USING       *,15
          L           10,A        A IN REGISTER 10
          BAL         6,DOUBLE    ADDRESS OF RET1 IN
                                  REGISTER 6
```

```
Name      Operation   Operand     Comment
RET1      ST          10,W
          L           10,B
          BAL         6,DOUBLE    ADDRESS OF RET2 IN
                                  REGISTER 6
RET2      A           10,W
          ST          10,W
          L           10,C
          BAL         6,DOUBLE    ADDRESS OF RET3 IN
                                  REGISTER 6
          A           10,W
          ST          10,W
          EOJ
DOUBLE    SLA         10,1        DOUBLE WHATEVER IS IN
                                  REGISTER 10
          BCR         15,6        RETURN TO THE ADDRESS
                                  GIVEN IN 6
          DS          0F
A         DS          F
B         DS          F
C         DS          F
W         DS          F
          END         BEGIN
```

Figure 6.5 gives the flowchart for this program, illustrating the multiple use of and return from the subroutine DOUBLE.

6.5 ADDRESS MODIFICATION AND LOOPS

If a particular sequence of instructions is to be repeated a number of times with changing operands, the standard technique is to set up the sequence in a programming loop. This obviously avoids the necessity of writing out the sequence a multiple number of times. In order for such a loop to work properly, the programmer must make sure that two basic mechanisms are available: First, there must be a way to change the operands so that the correct ones are always used. Second, there must be a way to determine when to stop repeating the particular loop and continue with subsequent instructions. At this point we shall have a look at an elementary way of doing this.

The general process of changing operands (irrespective of the particular technique used) is called *address modification.* In most instances, the type of situation calling for address modification is one in which a set of operations is to be performed on a group of similar operands (for example, fields of equal length). When this is the case, address modification is accomplished merely by successively adding a constant to an operand address each time the loop is executed.

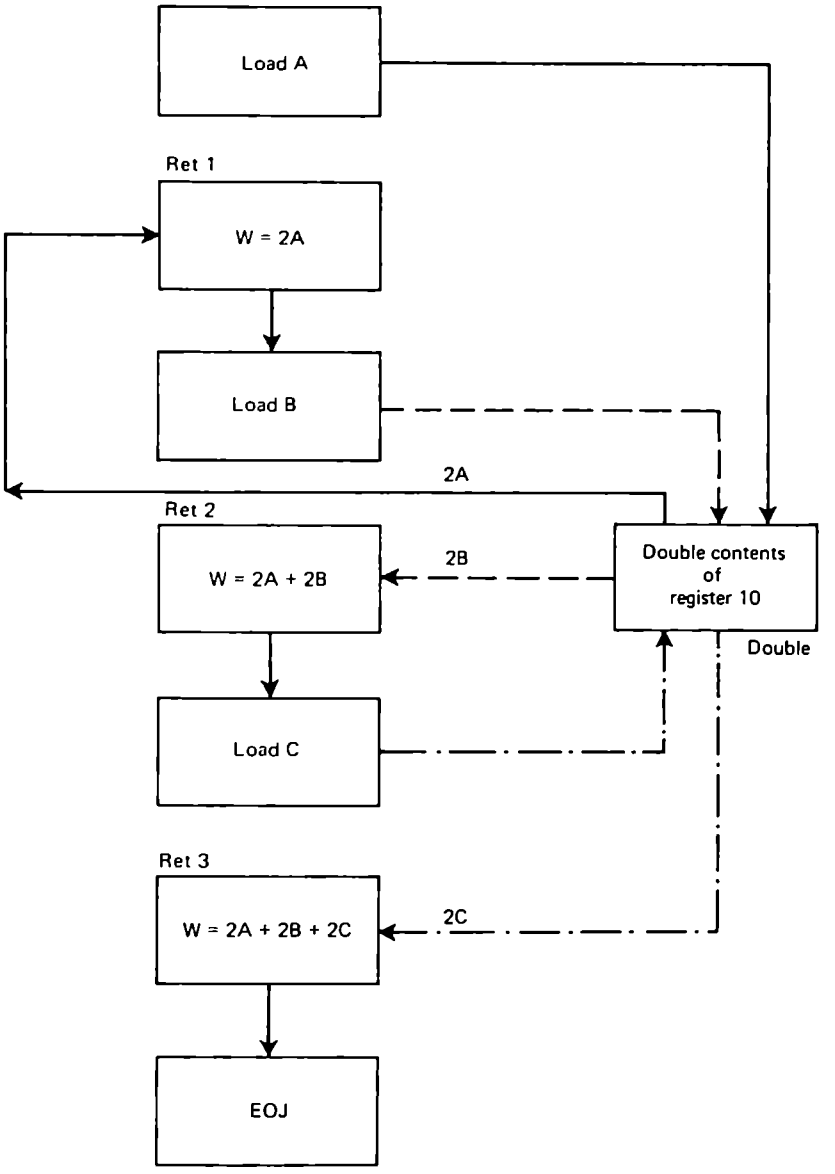


Figure 6.5

For example, if we had a collection of values stored in contiguous fullwords beginning with GROUP, the first value would be in GROUP, the second in GROUP+4, the third in GROUP+8, and so on. Thus if we arrange to add a value of four to the symbol GROUP each time through the loop, we shall produce the desired address modification.

Since the operand address portion of an instruction is in its rightmost bits, a simple way of modifying it is to add the desired constant directly

to the instruction. Suppose, for instance, that we wished to double each value in this collection of fullwords beginning in GROUP. Forgetting for the moment about how to turn this thing off, we could say something like:

```
LOOP  L     7,GROUP
      SLA   7,1
PUT   ST    7,GROUP
      L     8,LOOP
      A     8,C4
      ST    8,LOOP
      L     8,PUT
      A     8,C4
      ST    8,PUT

      .     .
some kind of test to determine
whether to recycle

      .     .
C4    DC    F'4'
```

Note that the second operand address in LOOP and PUT (the one that starts out as GROUP) is the item being modified.

Now we can consider the second required mechanism for such a loop. If the number of repetitions is known in advance (at any point before the loop is entered), it is possible to regulate the iterations by using the BCT (Branch on Count) instruction. This RX instruction has the general form

name BCT *n1,operand*

When this instruction is executed, it subtracts 1 from the contents of the register specified in *n1*. If that register is not zero after the subtraction, a branch is executed to the address given in *operand*. When the value in *n1* reaches zero, that branch is ignored and execution continues with the next sequential instruction.

Going back to our little example, we can now set up our loop to perform the doubling operation a predetermined number of times, each time on a different operand. We shall set it for five iterations:

```
      L     4,C5
LOOP  L     7,GROUP
      SLA   7,1
```

```
PUT     ST      7,GROUP
        L       8,LOOP
        A       8,C4
        ST      8,LOOP
        L       8,PUT
        A       8,C4
        ST      8,PUT
        BCT     4,LOOP
NEXT    further instruction

        .       .
C4      DC      F'4'
C5      DC      F'5'
```

The first time around, register 4 contains a value of 5, and the second operand in LOOP and PUT is GROUP. Then the operand in LOOP and PUT is changed from GROUP to GROUP+4. Reduction of the value in register 4 to 4 is followed by a branch back to LOOP and the process is repeated. GROUP+4 is changed to GROUP+8, register 4's value is reduced to 3, etc. When this modified address is GROUP+20, register 4 contains a 1. The BCT changes it to zero, inhibiting execution of the branch to LOOP. Instead, NEXT is executed, so that the last value to be doubled was the one in GROUP+16.

BCTR is an RR instruction which does the same thing as BCT, except that the second operand is a general register. It also provides a convenient way to reduce a register by 1 without branching. This is done by specifying a second operand of zero.

EXAMPLE 15

Compute $W = \sum_{i=1}^{100} Vi$, where each of the Vi's is a binary word, the first of which has an address of V.

Solution Procedure

1. Load 100 into a general register.
2. Set a general register to zero for accumulation of the sum.
3. Add V to the sum register.
4. Add 4 to the address of V in the add instruction.
5. If all 100 fields have been added, continue; otherwise, branch back to the add instruction.
6. Store the result in W.

Solution Program

```
Name      Operation   Operand      Comment
          START       8800
ADDER     BALR        15,0
          USING       *,15
          L           10,HNDRD     100 IN REGISTER 10
          SR          11,11        0 IN REGISTER 11
ADDON     A           11,V
          L           12,ADDON
          A           12,C4        OPERAND ADDRESS IN ADD
                                   IS MODIFIED
          ST          12,ADDON
          BCT         10,ADDON     BRANCH TO ADD UNLESS
                                   REG. 10 = 0
          ST          11,W         COMPLETED SUM IN W
          EOJ
          DS          0F
HNDRD     DC          F'100'
V         DS          100F
C4        DC          F'4'
W         DS          F
          END         ADDER
```

The flowchart is shown in Figure 6.6.

6.6 INDEXING

We can automate the process of address modification by using the technique of *indexing*. When the required modification is by a constant amount (as it was in the previous example) that constant can be placed in a general register (which then serves as an *index register*) and a mechanism can be set up for the starting address to be changed as part of the process of testing for completion of the iterations.

This technique requires the use of RX instructions in their full form. Recall (Chapter 3) that the effective address of an RX instruction's second operand is specified by a displacement, a base register and an index register. Up to this point, the indexing capability has not been used. We restricted ourselves to a base register (specified by USING) and a displacement (computed by the assembler).

The index register in an RX assembler statement is specified as a number enclosed in parentheses immediately following the second operand. For instance, the statement

```
A   12,X(7)
```

specifies that the value in register 12 is to be increased by the value in the address given by X modified by the contents of register 7.

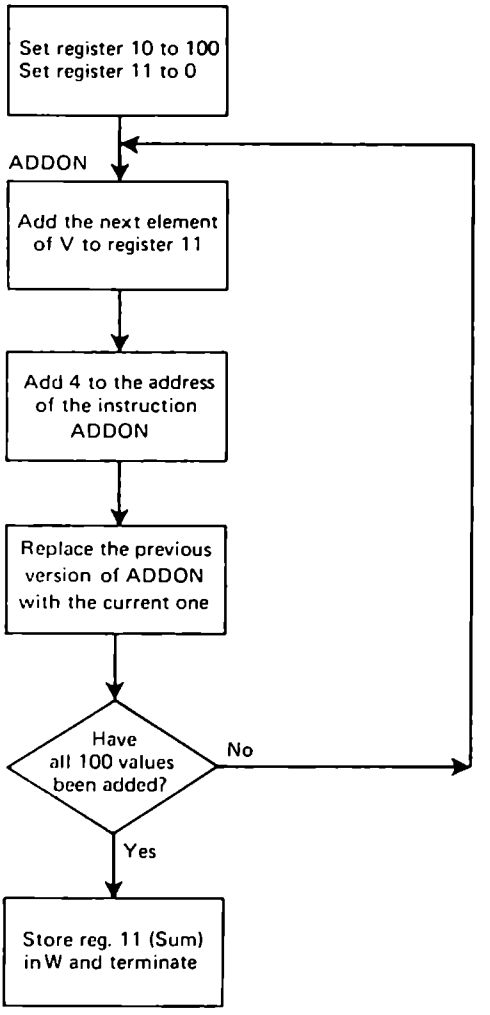

Set register 10 to 100
Set register 11 to 0
ADDON
Add the next element of V to register 11
Add 4 to the address of the instruction ADDON
Replace the previous version of ADDON with the current one
Have all 100 values been added?
No
Yes
Store reg. 11 (Sum) in W and terminate

Figure 6.6

Two instructions aid in the use of index registers: BXLE and BXH. The BXLE (Branch on Index Low or Equal) is an RS instruction with three operands. Its general form is

name BXLE *n*1,*n*2,*operand*

Although two registers (*n*1 and *n*2) are specified in the BXLE instruction, three are actually involved. The third of these is implied and must be the last in a series of three consecutive registers beginning with an odd-numbered one. (This series is often called an *odd-even-odd trio*). The BXLE operates as follows: The value in register *n*2 is added to the value in register *n*1 and the newly formed result (in *n*1) is compared with the contents of the odd-numbered register following *n*2. If the value in *n*1 is less than or equal to that in the third register, a branch is executed to the address speci-

fied by *operand*. Otherwise, when *n*1's value is greater, that branch is ignored and the next sequential instruction is executed.

This works out very nicely for setting up indexing loops. Register *n*1 is set at the starting value, *n*2 is equipped with the amount to be added each time the loop is repeated and the last of the odd-even-odd trio is given a value which represents the stopping point. With the registers initialized in this manner, the instructions in the loop can be written, followed by a BXLE. Each time the BXLE is reached, *n*1 will automatically be incremented by *n*2 and, as long as *n*1's value does not exceed the stopping point, the loop will be repeated.

EXAMPLE 16

Repeat the previous example, using the BXLE instruction.

Solution Procedure

1. Set register 10 to zero (we shall accumulate our sum there).
2. Registers 11, 12, and 13 will be our odd-even-odd trio. Set register 11 to zero.
3. Set register 12 (the increment register) to 4.
4. Set register 13 to the stopping value (396). V+396 is the last data item to be processed. Hence, when register 11 goes to 400, the branch will no longer execute.
5. Add N, indexed by register 11, to the accumulator (register 10).
6. Check for completion with a BXLE.
7. When looping is complete, store the final result in W.

Solution Program

```
Name    Operation   Operand        Comment
        START       5120
INDEX   BALR        15,0
        USING       *,15
        L           10,CZERO
        LM          11,13,CZERO    0 IN 11, 4 IN 12, 396 IN 13
LOOP    A           10,V(11)       SUM IS INCREASED BY NEXT V
        BXLE        11,12,LOOP     REPEAT LOOP TILL REG. 11>396
OUT     ST          10,W
        EOJ
        DS          0F
CZERO   DC          F'0'
CFOUR   DC          F'4'
CSTOP   DC          F'396'
W       DS          F
V       DS          100F
        END         INDEX
```

Although this sequence is only three machine instructions shorter than that required to get the same job done without indexing (Example 15), the effective savings in processing time is really much greater, since this reduction occurs each time the

loop is cycled. For something this small, the time saved is inconsequential. In more realistic situations, however, it can be considerable.

The BXH (Branch on Index High) instruction works the same way as BXLE, except that the branching criterion is reversed. This time it is presumed that *n*1 (the index value) is approaching the stopping value from the high side (that is, *n*2 is a negative constant and its (algebraic) addition to *n*1 actually decrements *n*1). As long as *n*1 exceeds the stopping value, the loop will repeat.

EXAMPLE 17

Repeat Example 16 using the BXH instruction.

Solution Procedure

Our procedure is the same as before, except that we will be starting at the other end of the collection of V's (effectively at V + 396) and working our way down through V (V + 0). Since the branch is stopped when the stopping point is reached (*not* when we fall below it), we shall have to set the stopping point at −4, to allow the first V to be processed.

Solution Program

```
*Name     Operation   Operand      Comment
          START       5140
INDEX1    BALR        15,0
          USING       *,15
          L           10,CZERO
          LM          11,12CBIG    396 IN 11,-4 IN 12,-4 IN 13
          L           13,DECR
LOOP1     A           10,V(11)
          BXH         11,12,LOOP1
OUT1      ST          10,W
          EOJ
          DS          0F
CZERO     DC          F'0'
CBIG      DC          F'396'
DECR      DC          F'-4'
W         DS          F
V         DS          1000F
          END         INDEX1
```

Although this was not the case here, there are many instances when there is a clear advantage in using BXLE rather than BXH, or vice versa.

PROBLEMS

1. Why is the ability to branch important in a digital computer?
2. What is meant by address modification? What are the advantages of this technique?
3. When does a programmer consider using subroutines? Why is special subroutine linkage required?

4. How does the use of an index register improve the operation of a program?
5. How is it possible for the BXLE and BXH instructions to have four operands when the RS format includes only three? How could a subroutine be used so as to simulate the effect of an instruction with, say, 10 operands?

Use the following definitions as needed:

Name	*Operation*	*Operand*
A	DS	F
B	DS	F
C	DS	F
D	DS	F
X	DS	100F

6. Write a program that sets B = C if A > 0 and sets B = D if A < 0.
7. Accumulate in A the sum of the positive fields in the array whose first address is X, and in B accumulate the sum of the negative fields.
8. Transform each quantity in the X array into its absolute value. Then accumulate in C the sum of the even integers, and accumulate in D the sum of the odd integers. [*Hint:* The low-order bit of an even integer is 0 and of an odd integer, 1.]
9. Set D to the value shown in the following table if both of the associated conditions are true.

Condition 1	*Condition 2*	*D*
A ≥ B	B < C	1
A ≥ B	B = C	2
A ≥ B	B > C	3
A < B	B ≤ C	4
A < B	B > C	5

10. Find the first quantity in the X array whose absolute value is greater than 1000. Place the contents of the field in General Register 5 and the address of the field in General Register 6.
11. Compute D = $(A^3 + B^3 + C^3)^3$ using a subroutine that calculates the third power of any quantity presented to it.
12. Set the first and second fields in the X array to 1 and compute the remaining quantities by the following rule: each quantity is equal to the sum of the preceding two quantities.
13. Compute 10! = 1*2*3*4*5*6*7*8*9*10 using an iterative loop.
14. Generalize the previous program; that is, given A, compute A!. Assume that A is a positive integer and that no overflow will occur.
15. Replace A, B, C, and D by A!, B!, C!, and D!, respectively, using a subroutine that calculates the factorial of a positive integer presented to it. Assume that A, B, C, and D are positive integers and that no overflow will occur.

Chapter 7

Decimal Operations

Previous chapters have emphasized that all digital computers are basically binary in nature and represent all information in terms of the binary digits 0 and 1. They have also stressed that binary numbers are represented directly, but that all other information—decimal numbers, hexadecimal numbers, alphabetic and special characters, and so forth—must be coded in terms of the binary digits. These facts make it appear that it is preferable to use the binary number system for computation since straightforward representation brings with it certain efficiencies.

While it is true that in digital computers the binary-arithmetic circuitry is simpler than it is for other modes of arithmetic, an attribute that is translatable into increased speed, there are other factors to consider in addition to computation. In particular, three of these are the input/output requirements, the nature of the program, and the background of the programmer. All should be considered in deciding what computational mode is most desirable for a particular problem program.

With respect to the first factor, input data and output results are ordinarily expressed in the decimal number system. To use binary arithmetic requires the conversion of input data from decimal to binary format before computation can begin, and the conversion of results from binary to decimal format before the desired output can be produced. The use of decimal arithmetic, on the other hand, requires no such conversion. In regard to the second factor, there are applications in which the performance of calculations within the computer should simulate as closely as possible the same calculations

as they would be performed manually. Here too the use of decimal arithmetic is the more desirable. As to the third factor, everyone learns the decimal number system first, and like one's mother tongue, it is the system in which one thinks. Other number systems, in comparison, retain a certain unfamiliarity, no matter how well they are learned or how easy they are to use. This factor, then, also deserves consideration in deciding the computational mode for a problem program.

If a computer provides no choice of arithmetic type, then the programmer need not concern himself with the selection of operating mode, being compelled instead to force all types of programs into the same structure, whether or not it is suitable. If a computer provides more than one choice of arithmetic mode, as does System/360 in providing three, the programmer is then in a position to apply the full versatility of the computer to the problems he wishes to solve. In this chapter the decimal arithmetic instructions and some related instructions are discussed. They are of special usefulness where variable-length data fields are to be processed and where number base conversions are time-consuming in relation to the computations required.

7.1 ADDITION AND SUBTRACTION

Since all decimal operations are designed to be performed on variable-length fields, the instructions are of the SS type. The basic format is

label *operation code* *operand*1,*operand*2

where *operand*1 and *operand*2 represent the symbolic storage addresses of the two fields to be manipulated. The fact that the instruction specified by the operation code is one of the decimal instructions causes the assembler (and, in turn, the processing circuitry) to assume that the operands are in the packed-decimal format.

Note that the basic format does not include a length specification for the operands. This is usually not necessary, since the length is available to the assembler through the DS or DC statements which allocated storage for the particular fields. Should the occasion arise for an explicit length definition, it can be given, enclosed in parentheses, immediately following its respective operand symbol:

label *operation code* *operand*1(4),*operand*2(2)

7.1.1 DC's and DS's for Packed-Decimal Fields

It will be helpful to review the mechanism for defining constants in packed-decimal format and allocating storage for such purposes. The latter operation can be dispensed with quickly. Since packed-decimal fields

must be an integral number of bytes in length, an appropriate number of bytes is allocated by a DS with a character designation:

```
name    DS      CLn
```

where *n* is the desired number of bytes to be reserved. Incidentally, since all decimal operations are byte-oriented, there is no necessity for concern about halfword, word, or doubleword alignment.

Both the P and X specifications can be used to generate packed-decimal constants. In either case, all packed-decimal constants must be signed. Positive constants defined with the P specification need not have an explicit sign included, since the assembler will automatically place a plus sign indicator (1100) in the rightmost half-byte. Thus

```
CONST   DC      P'50'
```

and

```
CONST   DC      P'+50'
```

will both cause a two-byte field to be defined under the name CONST, with a value of

0000 ¦ 0101	0000 ¦ 1100

being stored in it. A specification of

```
CONST   DC      P'-50'
```

instead, would cause the stored value to be

0000 ¦ 0101	0000 ¦ 1101

The programmer can control the number of bytes to be allocated by using the L specification. For example, the statement

```
X       DC      PL3'-50'
```

will place the value

0000 ¦ 0000	0000 ¦ 0101	0000 ¦ 1101

in a three-byte field with the first (leftmost) byte having the symbolic address X.

When using the X specification to define a packed-decimal value the programmer must always specify the sign. Since the plus sign is 1100, its corresponding hexadecimal value (C) is placed at the end of all positive numbers in DC statements using the X specification. Thus the statement

```
CONST   DC      X'50C'
```

will produce the same results as

```
CONST   DC      P'+50'
```

Similarly, the negative sign (1101) is specified by a D, thus making the statements

```
CONST   DC      X'50D'
```

and

```
CONST   DC      P'-50'
```

equivalent. The length specifications can be used to control the number of bytes.

7.1.2 Decimal Addition

Addition is performed by the AP (Add Decimal) instruction:

name AP *operand*1,*operand*2

the contents of *operand*2 are added to *operand*1. As an example, suppose we defined A and B as

```
A       DC      PL3'20744'
B       DC      PL3'695'
```

and said

```
AP      A,B
```

the operation could be visualized as

2	0	7	4	4	C	
0010	0000	0111	0100	0100	1100	A
0	0	6	9	5	C	
0000	0000	0110	1001	0101	1100	B

after Addition:

2	1	4	3	9	C	
0010	0001	0100	0011	1001	1100	A

As a result of an AP operation, the condition code is set as follows:

Condition Code	*Table Value*	*Produced When*
00	8	Result = 0
01	4	Result < 0
10	2	Result > 0
11	1	Overflow

It is up to the programmer to make sure that the field for a decimal sum will be large enough to accommodate it. This usually implies the allocation of an extra byte in the first operand so that overflow is prevented.

The ZAP (Zero and Add Decimal) instruction can be used to load a packed decimal value into a field. Its general form is

name ZAP *operand1,operand2*

The field specified by *operand1* is set to zeros and filled with the contents of *operand2*, starting with the low-order half-byte (the sign) and working to the left.

ZAP sets the condition code as follows:

Condition Code	*Table Value*	*Produced When*
00	8	*Operand2* = 0
01	4	*Operand2* < 0
10	2	*Operand2* > 0

7.1.3 Decimal Subtraction

Subtraction is handled with the SP (Subtract Decimal) instruction. Its operation parallels that of the AP instruction. Thus

name SP *operand1,operand2*

will cause the contents of *operand2* to be subtracted from *operand1* with the resulting difference being formed in *operand1*. The condition code is set as follows:

Condition Code	*Table Value*	*Produced When*
00	8	Result = 0
01	4	Result < 0
10	2	Result > 0
11	1	Overflow

A packed-decimal field can be set to zero by subtracting it from itself.

EXAMPLE 18

L, M, and N are three-byte packed decimal fields containing values of 312, −175, and −67, respectively. Write a program which computes L + M − N and stores it in the four-byte field W.

Solution Procedure

1. Clear the field W to zero and add L to it.
2. Add M to W.
3. Subtract N from W to obtain the required result.

Solution Program

```
Name      Operation  Operand      Comment
          START      6600
DECADD    BALR       15,0
          USING      *,15
          ZAP        W,L          L IN W
          AP         W,M          L+M IN W
          SP         W,N          L+M-N IN W
          EOJ
L         DC         PL3'312'
M         DC         PL3'-175'
N         DC         XL3'00067D'  JUST FOR VARIETY
W         DS         CL4
          END        DECADD
```

7.2 DATA CONVERSION

Although decimal operations are convenient to use and conserve storage (because of packing) and binary operations are fundamentally faster and simpler than those for other forms, neither form is convenient for submitting input. As we saw in Chapter 2, it is more natural to present numeric data in EBCDIC (zoned-decimal) form. Consequently, some mechanism must be provided to convert such input, which cannot be processed, to packed-decimal or binary form.

7.2.1 Conversion Between Packed- and Zoned-Decimal Formats

The reader will recall (Sec. 2.3.1) that numerical input data, when recorded without an explicit decimal point and with the sign superimposed on the low-order digit, will be stored automatically in zoned-decimal format. For example, a value keypunched as $436\bar{9}$ will appear internally as

F	4	F	3	F	6	D	9
1111	0100	1111	0011	1111	0110	1101	1001

The Assembler Language programmer can convert zoned-decimal data to packed-decimal format with the Pack Data (PACK) statement, whose general form is

name PACK *operand*1,*operand*2

When the machine-language instruction corresponding to this statement is executed, the contents of *operand*2 will be packed and stored in *operand*1. As with other SS type instructions, the operand lengths may be stated or

implied. If *operand*2 is m bytes long, the minimum required length for *operand*1 is $(m + 1)/2$, rounded upward to the nearest byte. For instance, our value of 4369 will be converted as shown:

F	4	F	3	F	6	D	9
1111	0100	1111	0011	1111	0110	1101	1001

zoned decimal

0	4	3	6	9	D
0000	0100	0011	0110	1001	1101

packed decimal

If the specified length of the receiving (packed) field exceeds that required for the incoming data, the appropriate number of high-order zeros will be filled in as part of PACK's execution. Neither operand may exceed 16 bytes. It is possible to pack a field into itself.

Conversion in the other direction is achieved by means of the Unpack (UNPK) instruction. As with the PACK instruction, *operand*1 is the receiving field. For example, if A is a three-byte packed decimal field containing

1	7	8	0	5	C
0001	0111	1000	0000	0101	1100

↑
Sign

and B is a seven-byte field, the statement

```
UNPK   B,A
```

the resulting contents of B will be

F	0	F	0	F	1	F	7	F	8
1111	0000	1111	0000	1111	0001	1111	0111	1111	1000

F	0	C	5
1111	0000	1100	0101

↑
Sign

A field not exceeding 16 bytes may be unpacked with one instruction. The operands in an UNPK instruction may not be the same (a field cannot be unpacked into itself).

EXAMPLE 19

P and Q are four-byte zoned-decimal fields containing respective values of 206 and −48. Write a program that computes 2P−Q and stores the result as a packed-decimal value in the four-byte field R and as a zoned-decimal value in the five-byte field S.

Solution Procedure

1. Convert P and Q to packed decimal.
2. Add P to itself and subtract Q.
3. Unpack the result into S.

Since we are not yet concerned with input, we shall generate the zoned-decimal values of P and Q as hexadecimal constants.

Solution Program

```
Name      Operation  Operand   Comment
          START      3600
CHANGE    BALR       15,0
          USING      *,15
          PACK       R,P       P IS IN R IN
                               PACKED FORMAT
          PACK       P,Q       Q IS IN P IN
                               PACKED FORMAT (WE
                               ARE SAVING STORAGE
                               BY USING P INSTEAD
                               OF A SEPARATE PLACE)
          AP         R,R       2*P IS IN R
          SP         R,P       2*P-Q IS IN R
          UNPK       S,R       2*P-Q IN ZONED FORMAT
                               IS IN S
          EOJ
P         DC         XL4'F0F2F0C6'
Q         DC         XL4'F0F0F4D8'
R         DS         CL4
S         DS         CL5
          END        CHANGE
```

Before Processing:

P	F 1111	0 0000	F 1111	2 0010	F 1111	0 0000	C 1100	6 0110

Q	F 1111	0 0000	F 1111	0 0000	F 1111	4 0100	D 1101	8 1000

After Processing: (Q's value)

P	0 0000	0 0000	0 0000	0 0000	0 0000	4 0100	8 1000	D 1101

(unchanged)

	F	0	F	0	F	4	D	8
Q	1111	0000	1111	0000	1111	0100	1101	1000

(2P-Q in packed format)

	0	0	0	0	4	6	0	C
R	0000	0000	0000	0000	0100	0110	0000	1100

(2P-Q in zoned format)

	F	0	F	0	F	4	F	6	C	0
S	1111	0000	1111	0000	1111	0100	1111	0110	1100	0000

7.2.2 Conversion Between Decimal and Binary Formats

The Convert to Binary (CVB) and Convert to Decimal (CVD) instructions mediate between packed-decimal and binary data formats. Both are RX instructions in which the first operand is a general register and the second is the address of a doubleword. The register always contains the data in binary format. Thus the instruction

```
CVB     3,DBLDBL
```

would convert the contents of DBLDBL (which we presume to be a doubleword) to binary form and place the converted value in register 3. Conversely, the statement

```
CVD     3,DBLWRD
```

will convert register 3's contents to packed decimal and place the result in DBLWRD.

EXAMPLE 20

A and B are four-byte zoned-decimal fields containing values of 132 and −16 respectively. Write a program that computes A − B in binary format and stores the result as a binary halfword in C, and a four-byte zoned-decimal value in D. Use the symbol DD for the doubleword working area.

Solution Procedure

1. Pack A into DD and convert to binary from there.
2. Pack B into DD and convert to binary from there.
3. Calculate A − B and store in C.
4. Convert A − B to packed decimal and store in DD.
5. Convert D to zoned decimal and store in D.

Solution Program

```
Name      Operation  Operand      Comment
          START      6400
ALTER     BALR       15.0
          USING      *,15
          PACK       DD,A         132 IS IN DD IN
                                  PACKED DECIMAL
STAGE1    CVB        5,DD         132 IS IN REGISTER 5
                                  IN BINARY
          PACK       DD,B         -16 IS IN DD IN
                                  PACKED DECIMAL
STAGE2    CVB        7,DD         -16 IS IN REGISTER 7
                                  IN BINARY
          SR         5,7          148 IS IN REGISTER 5
                                  IN BINARY
          STH        5,C          148 IS IN C IN BINARY
          CVD        5,DD         148 IS IN DD IN
                                  PACKED DECIMAL
          UNPK       D,DD+6(2)    UNPACK THE LAST TWO
                                  BYTES OF DD INTO D
          EOJ
A         DC         XL4'F0F1F3C2'
B         DC         XL4'F0F0F1D6'
          DS         0D
DD        DS         D
D         DS         CL4
C         DS         H
          END        ALTER
```

Before Processing:

A

F	0	F	1	F	3	C	2
1111	0000	1111	0001	1111	0011	1100	0010

B

F	0	F	0	F	1	D	6
1111	0000	1111	0000	1111	0001	1101	0110

Immediately After STAGE1:

DD

0	0	0	0	0	0	0	0	0	0
0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

0	0	1	3	2	C
0000	0000	0001	0011	0010	1100

Register 5

00000000	00000000	00000000	00000000	00000000	00000000

00000000	10000100

Immediately After STAGE2:

DD

0	0	0	0	0	0	0	0	0	0
0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

0	0	0	1	6	D
0000	0000	0000	0001	0110	1101

Register 7

11111111	11111111	11111111	11111111	11111111	11111111

11111111	11110000

After Conclusion:

Register 5

00000000	00000000	00000000	00000000	00000000	00000000

00000000	10010100

C

00000000	10010100

DD

0	0	0	0	0	0	0	0	0	0
0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

0	0	1	4	8	C
0000	0000	0001	0100	1000	1100

D

F	0	F	1	F	4	C	8
1111	0000	1111	0001	1111	0100	1100	1000

7.3 MULTIPLICATION AND DIVISION

As is the case with binary values, decimal multiplication and division can each be performed with single instructions. However, the programmer must do a little more preparation, since he must make sure that ample storage is allocated for the results.

7.3.1 The MP Instruction

Multiply Decimal (MP) is an SS instruction which operates on two packed-decimal fields. The first operand is considered to be the multiplicand and the second the multiplier, with the product being developed in the multiplicand. Because the number of digits in a product is either the sum of the numbers of digits in the multiplier and multiplicand or one less than said sum, it is necessary to have at least as many high-order zeros in the multiplicand as there are digits in the multiplier, so that there is enough space provided for the product. A corollary of this requirement is that the size of the multiplier must always be less than that of the multiplicand. Since the largest field size allowed in the decimal format is 31 digits plus sign (16 bytes) a second corollary is that the multiplier size cannot be larger than 15 digits plus sign. If any of these rules is broken, MP cannot be executed properly.

The actual multiplication is usually preceded by a ZAP statement which zeros the product area and places the multiplicand in its low-order bytes. The MP instruction does not affect the condition code.

7.3.2 The DP Instruction

Divide Decimal (DP) operates in much the same way as MP. The first operand is considered to be the dividend, the second is treated as the divisor, and the resulting quotient together with its remainder, is developed in the dividend field. Analogous to multiplication, the dividend length may be the maximum allowable field length (16 bytes) but the divisor's length is restricted to a maximum of eight bytes. At the completion of the division process, the remainder, equal in length to the divisor and carrying the same sign as the dividend, will be in the low-order bytes of the dividend field. The quotient and its sign will occupy the remaining high-order bytes of the dividend field. Division is usually preceded by a ZAP statement which sets up the dividend.

EXAMPLE 21

X is a three-byte zoned-decimal value of 309, Y is a packed-decimal value of 24, and Z is a binary halfword containing 7. Write a program to calculate XY/Z and store the resulting quotient as a zoned-decimal value in the six-byte field Q and the remainder as a four-byte packed-decimal in field R.

Solution Procedure

1. Convert X and Z to packed-decimal format.
2. Multiply X by Y and then divide by Z.
3. Convert X*Y/Z to zoned decimal and store in Q.

We shall need an area for developing the product and quotient. Since X and Y have a total length of five digits, this should be sufficient for the product. If we are to use that same field for handling the subsequent division operations, we must make it long enough for six digits, so that dividend requirements are fulfilled. We will call this field WORK. We shall also have to set aside a doubleword (which we will call BIN) for handling the binary-to-decimal conversion of Z.

Solution Program

```
Name    Operation   Operand          Comment
        START       5500
MOIV    BALR        15,0
        USING       *,15
        PACK        X,X              308 IN X IN PACKED
                                     FORMAT
        LH          6,Z              7 IN REGISTER 6
        CVD         6,BIN            7 IN DOUBLEWORD
                                     BIN IN PACKED
                                     FORMAT
        ZAP         WORK,X           308 IN WORK
PART1   MP          WORK,Y           308*24 IN WORK
        DP          WORK,BIN+7(1)    DIVIDE BY LAST BYTE
                                     IN BIN
        UNPK        Q,WORK(3)        QUOTIENT IN Q
        ZAP         R,WORK+2(1)      REMAINDER IN R
        EOJ
X       DC          XL3'F3F08C'
Y       DC          XL2'024C'
        DS          0D
BIN     DS          D
R       DS          CL4
Z       DC          H'7'
Q       DS          CL6
WORK    DS          CL4
        END         MOIV
```

Before Processing:

X

F	3	F	0	C	9
1111	0011	1111	0000	1100	1000

Y

0	2	4	C
0000	0010	0100	1100

Z

00000000	00000111

Immediately After PART1:

X	0 0000	0 0000	3 0011	0 0000	9 1000	C 1100					
BIN	0 0000	0 0000	0 0000	0 0000	0 0000	0 0000	0 0000	0 0000	0 0000	0 0000	
	0 0000	0 0000	0 0000	0 0000	7 0111	C 1100					
WORK	0 0000	0 0000	0 0000	7 0111	4 0100	1 0001	6 0110	C 1100			

At Conclusion:

WORK	0 0000	1 0001	0 0000	5 0101	9 1001	C 1100	3 0011	C 1100			
Q	0 0000	0 0000	0 0000	0 0000	F 1111	1 0001	F 1111	0 0000	F 1111	5 0101	
	C 1100	9 1001									
R	0 0000	0 0000	0 0000	0 0000	0 0000	0 0000	3 0011	C 1100			

Now that we have seen how the basic packed-decimal division operation works, we shall briefly mention some of its further properties which the programmer must take into account when planning a DP. In general, if a dividend is *m* digits long (not counting sign or high order zeros) and a divisor is *n* digits long (measured the same way) the number of digits in the quotient

(let us call that q) will be

$$q = m - n + 1$$

If the dividend field is not long enough for the quotient and remainder, division will not take place and an interrupt will occur. Consequently, the programmer must make sure that the size of the dividend field is sufficient to avoid this. For example, he must account for the fact that, after division has taken place, there are separate signs for quotient and remainder. It turns out that since the dividend field must ultimately accommodate a quotient of $(m - n + 1)$ digits plus its sign (or $m - n + 2$) and a remainder of n digits plus its sign (or $n + 1$), a minimum field length for the dividend would be $m - n + 2 + n + 1$ (or $m + 3$) in half-bytes.

Length considerations may also be required to account for a range in the number of places in the divisor values to be processed by a particular DP instruction. Also, the programmer may wish to increase the number of decimal places in a quotient by providing additional space in the dividend field. This latter consideration is discussed in Sec. 7.4.6.

7.4 INTERNAL MOVEMENT OF DATA

Although the ZAP instruction represents a way of moving information from one field to another, its flexibility is rather limited. There are numerous occasions calling for partial alteration of a field's contents or various types of internal shifting. The Assembler Language programmer can perform these manipulations fairly conveniently using a versatile array of Move instructions.

7.4.1 The MVC Instruction

The Move Character (MVC) instruction allows the transfer of single bytes (or fields up to 256 bytes in length) between designated locations. Its general form

name MVC *operand1,operand2*

causes the contents of *operand2* to be duplicated in *operand1*. The operation proceeds from left to right, starting with the high-order byte of each operand. *Operand2*'s contents are unaffected and nothing happens to the uninvolved bytes of *operand1*. The number of bytes that are duplicated (moved) are dictated by the length that is specified or implied in *operand1*. The condition code is not set. We can see this action more clearly in some examples.

Let us suppose that A and B are zoned-decimal constants defined as

```
A    DC    XL6'F4F3F0F1F6C5'
B    DC    XL4'F2F9F7D8'
```

The statement

```
MVC     A,B
```

will reproduce the contents of B in A, B+1 in A+1, B+2 in A+2, and B+3 in A+3. A+4 and A+5 will remain unchanged, as will all of field B. Thus A's final value will be F2F9F7D8F6C5. Had we written

```
MVC     A(3),B
```

A's contents would have been changed to F2F9F7F1F6C5. Table 7.1 gives some additional examples for these initial A and B values.

Both operands in an MVC statement may be parts of the same field. However, the programmer must be aware of MVC's action in order to insure proper use of this form. For example, if A is defined as

```
A       DC      XL6'F3F2F4F5F1C7'
```

and we say

```
MVC     A(3),A+1
```

A+1 will be produced in A, A+2 in A+1, and A+3 in A+2, thus changing A's value to F2F4C5F5F1C7. However, if we had written

```
MVC     A+1(3),A
```

A's value would be duplicated in A+1, then A+1's value (now the same as A's) would be placed in A+2, and so on. Consequently, A's new value would be F2F2F2F2F1C7. Thus we can use MVC for shifting to the left but not to the right. On the other hand, it is very convenient for filling a long field with a particular character.

7.4.2 The MVI Instruction

The Move Immediate (MVI) instruction can be used to place a single self-defined character into the byte designated by the first operand. Thus

```
MVI     PLACE,C'b'
```

TABLE 7.1 EXAMPLES OF MVC USAGE
(A=F4F3F0F1F6C5, B=F2F9F7D8)

Statement		Resulting Value of A
MVC	A+4(2),B	F4F3F0F1F2F9
MVC	A(3),B+1	F9F7D8F1F6C5
MVC	A(1),B+3	D8F3F0F1F6C5
MVC	A+5(1),B+3	F4F3F0F1F6D8

will place a single blank in the byte named PLACE. Of course, self-defined constants can also be specified in other forms. For example,

```
MVI   A,X'F2'
MVI   B+2,26
MVI   LOOK,C'W'
```

are all valid.

EXAMPLE 22

A packed-decimal field named FIELD and a byte named BYTE are in memory. Set BYTE to −1, 0, or 1 according to whether FIELD is negative, zero, or positive, respectively.

Solution Procedure

1. Set the condition code according to the sign of FIELD.
2. If FIELD<0, set BYTE = −1; if FIELD = 0, set BYTE = 0; if FIELD>0, set BYTE = 1.

Solution Program

```
Name    Operation   Operand
        START       6500
SET     BALR        15,0
        USING       *,15
        ZAP         FIELD,FIELD
        BC          4,NEG
        BC          8,ZERO
        MVI         BYTE,X'1C'
        BC          15,OVER
NEG     MVI         BYTE,X'1D'
        BC          15,OVER
ZERO    MVI         BYTE,X'0C'
OVER    EOJ
FIELD   DS          CL6
BYTE    DS          C
        END         SET
```

7.4.3 Movement of Half-Bytes

Two instructions are used to duplicate the contents of selected half-bytes. Both of them are very similar to the MVC instruction in that they are of the SS type and work from left to right.

MVN (Move Numeric) will duplicate the contents of the rightmost half of each byte in the second operand in the corresponding half-bytes of the first operand. Since the control circuitry makes no distinction between

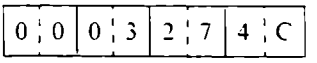
zoned-decimal formats for this instruction. MVN may be used with either type. For example, we shall replace the sign in a packed-decimal field A. Let us suppose that A and B are defined as

```
A    DC    PL4'3274'
B    DC    XL1'C6'
```

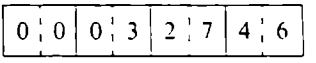
Then the statement

```
MVN    A+3(1),B
```

will change A from



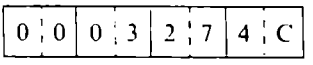
to



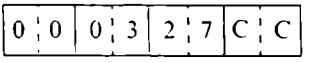
MVZ (Move Zones) does the same thing using the leftmost four bits of each byte. Thus, using A and B above, the statement

```
MVZ    A+3(1),B
```

will change A from



to



Note that A will no longer be accepted as a packed decimal field.

As with the MVC, the number of half-bytes moved is controlled by the length of the first operand. Both operands may be parts of the same field, but the programmer must exercise the same care as in MVC.

7.4.4 The MVO Instruction

The MVO (Move with Offset) operates somewhat differently from the other Move instructions discussed thus far. Although it is an SS instruction, which duplicates a specified number of bytes, it operates from right to left. (Recall that MVC, MVN, and MVZ operate from left to right). Furthermore, there is a shift that occurs automatically as part of the operation. Referring to the general format

```
name    MVO    operand1,operand2
```

the bytes specified in *operand*2 are reproduced in *operand*1 offset to the left by four bits. Unlike MVC, MVN, and MVZ, the length in *operand*1 is *not*

controlling. Let us start with a straightforward example. Suppose A is four bytes long and initially contains

6 2	7 1	4 5	9 C

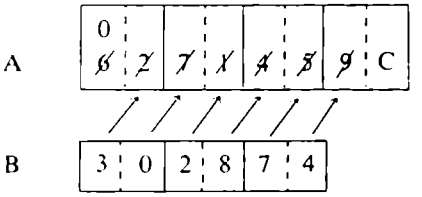
and B, three bytes long, contains

3 0	2 8	7 4

The statement

```
MVO     A,B
```

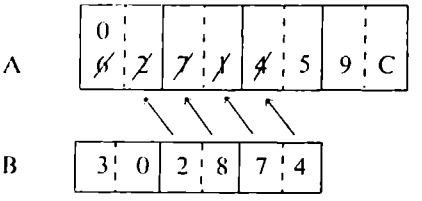
will cause the following replacements:



The 6 in the leftmost half-byte of A is automatically replaced by zero, producing a new A of | 0 3 | 0 2 | 8 7 | 4 C |. Had we written

```
MVO     A(3),B+1(2)
```

the resulting operation would be as follows:



thereby changing A to

0 2	8 7	4 5	9 C

EXAMPLE 23

Z is a seven-byte zoned-decimal field containing a value of 2468135. Write a program that produces

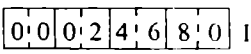
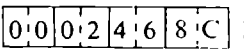
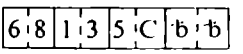
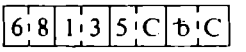
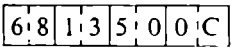
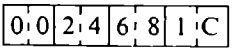
1. A packed-decimal field B with a value of 0002468.
2. A packed-decimal field C with a value of 6813500.
3. A new Z value of 0034681.

Solution Procedure

1. Pack Z into a four-byte field Z1.
2. Bring the numeric portion into B with an MVO.

3. Bring the sign into B with an MVN.
4. Bring the required value into C with an MVC.
5. Place C's sign in the proper position.
6. Place zeros in the proper half-bytes.
7. Shift Z1's sign two places left.
8. Shift Z1 two places to the right with a ZAP instruction.

Solution Program

Name	*Operation*	*Operand*	*Comment*
	START	4096	
SHIFF	BALR	15,0	
	USING	*,15	
	PACK	Z1,Z	IN Z1
	MVO	B,Z1(2)	 IN B
	MVN	B+3(1),Z1+3	 IN B
	MVC	C(3),Z1+1	 IN C
	MVN	C+3(1),C+2	 IN C
	MVO	C+3(1),ZERO	 IN C
	MVN	Z1+2(1),Z1+3	IN Z1
	ZAP	Z1,Z1(3)	 IN Z1
	UNPK	Z,Z1	
	EOJ		
Z	DC	XL7'F2F4F6F8F1F3C5'	
Z1	DS	CL4	
B	DS	CL4	
C	DS	CL4	
ZERO	DC	X'00'	
	END	SHIFF	

EXAMPLE 24

A zoned-decimal field six bytes long has its sign in the byte with the high-order digit. Move the sign to the byte with the low-order digit, and set to zero the bits from which it was moved.

Solution Procedure

1. Move the sign to the low-order byte.
2. Replace the original location of the sign with zeros.

Solution Program

```
Name    Operation  Operand        Comment
        START      5000
ZONES   BALR       15,0
        USING      *,15
        MVZ        ZDF+5(1),ZDF   ZDF HAS TWO SIGNS
        MVZ        ZDF(1),ZERO    ZERO OUT HIGH-ORDER
                                  SIGN
        EOJ
ZDF     DS         CL6
ZERO    DC         X'00'
        END        ZONES
```

7.4.5 Rounding

One of the most important uses of the Move instructions is in rounding results of decimal computations. The rule followed in this operation is the same one used for manual calculations: if the decimal place following the one to be rounded is 0–4, no change is made; if it exceeds 4 it is increased by 1 (if the number is positive) or decreased by 1 if the number is negative. Several illustrations are shown in Table 7.2

TABLE 7.2 ILLUSTRATIONS OF RULE FOR ROUNDING (ASSUME ROUNDING TO TWO PLACES)

Number	Rounded to
12.573	12.57
390.088	390.09
−2.976	−2.98
−31.003	−31.00

This process can be simplified on computers by reducing the number of decisions and simply adding 5 to the place following the one to be rounded (or subtracting 5 if the number is negative) and then discarding that place. Thus 390.088 is rounded to two places by 390.088 + .005 = 390.093.

EXAMPLE 25

A packed-decimal field named PDF consists of nine digits and a sign. The decimal point, though not present explicitly, is as shown in the following representation, where D and S stand for digit and sign positions respectively.

DDDDD.DDDDS

Round off PDF to the nearest hundredth, then place the result in a four-byte field named RESULT.

Solution Procedure

1. Set the condition code according to the sign of the field being rounded.
2. If the field is positive or zero, add 5 into the thousandths' place; if negative, subtract 5 from the thousandths' place.
3. Move the sign to the thousandths' place.
4. Move the rounded field to RESULT.

Solution Program

```
Name     Operation  Operand          Comment
         START      5000
ROUND    BALR       15,0
         USING      *,15
         ZAP        PDF,PDF          SET CONDITION
                                     CODE FOR PDF
         BC         4,NEG
         AP         PDF,FIVE
         BC         15,OVER
NEG      SP         PDF,FIVE
OVER     MVN        PDF+3(1),PDF+4
         MVC        RESULT,PDF
         EOJ
PDF      DS         CL5
RESULT   DS         CL4
FIVE     DC         X'50C'
         END        ROUND
```

EXAMPLE 26

Repeat the previous example, this time rounding to the nearest tenth rather than to the nearest hundredth. The rounded result is to be placed, right-justified, into RESULT.

Solution Procedure

1. Set the condition code according to the sign of the number to be rounded.
2. If the number is positive or zero, add 5 to the hundredths' place; otherwise, subtract 5.

3. Move the sign to the sign position of RESULT.
4. Move the six digits to RESULT, offsetting them so as to be adjacent to the sign position.

Solution Program

```
Name      Operation   Operand              Comment
          START       6500
TENTH     BALR        15,0
          USING       *,15
          ZAP         PDF,PDF              SET CONDITION
                                           CODE FOR PDF
          BC          4,NEG
          AP          PDF,FIVE             ROUND POSITIVE
                                           VALUE
          BC          15,OVER
NEG       SP          PDF,FIVE             ROUND NEGATIVE
                                           VALUE
OVER      MVN         RESULT+3(1), PDF+4   SHIFT SIGN
          MVO         RESULT,PDF(3)
          EOJ
PDF       DS          CL5
RESULT    DS          CL4
FIVE      DC          X'500C'
          END         TENTH
```

7.4.6 Increased Precision in Division Operations

As was seen previously (Sec. 7.3.2) a decimal division operation involving an m-digit dividend and an n-digit divisor is designed to produce an $(m - n + 1)$-digit quotient and n-digit remainder. It is often desirable to increase the number of decimal places in such results, for which purpose the Move operations can be conveniently used.

The general technique is to allocate additional low-order bytes for the result field and locate the dividend in it so that the appropriate number of places overhang. Since we cannot add an odd number of half-bytes directly, some additional manipulation is necessary to produce this type of increase in the number of places.

EXAMPLE 27

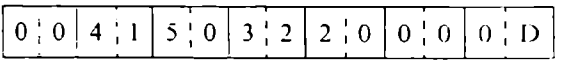
A seven-digit dividend X consists of a five-digit integer and two decimal places. A divisor Y is five digits long, one of which is to the right of an assumed decimal point. Write a program that will perform the division X/Y to produce a quotient

with four decimal places. For purposes of this illustration X and Y have values of −41503.22 and 2011.4 respectively. Store the quotient in Z and the remainder in R.

Solution Procedure

Unless we do something about it, the DP operation will produce a quotient with a number of decimal places equal to the difference between the number in the dividend and that in the divisor. In our particular case, that value is 2 − 1 (or 1 place to the right of our assumed decimal point). Consequently, we shall need four more places to the right of our dividend (three to fulfill our requirements and an extra one to use for rounding). Our basic dividend field size, then, would be 7 + 3 or 10 half-bytes, to which we shall add four places, thus defining its length as 14 half-bytes, or seven bytes. We shall call this field WORK.

1. We must position X in WORK so that the four extra places are to the right. That is, we want WORK to look as follows before we divide:



2. After we have divided, we shall round the quotient to four places by adding five to the extra (fifth) place and truncating.

Solution Program

Name	*Operator*	*Operand*	*Comment*
	START	6500	
PLACE	BALR	15,0	
	USING	*,15	
	ZAP	WORK(5),X	WORK HAS 0 0 4 1 5 0 3 2 D 0 0 0 C
	MVN	WORK+6(1),WORK+4	0 0 4 1 5 0 3 2 2 0 0 0 D
	MVO	WORK+5(1),CZERO	0 0 4 1 5 0 3 2 2 0 0 0 D
	DVP	WORK,Y	2 0 6 3 8 9 2 D 0 6 3 1 2 D
	SP	WORK(3),C5	2 0 6 3 8 9 7 D 0 6 3 1 2 D
	MVO	WORK(4),WORK(3)	0 2 0 6 3 8 9 D 0 6 3 1 2 D
	MVC	Z,WORK	
	MVC	R,WORK+4	
	EOJ		

```
Name    Operator  Operand
WORK    DC        PL7'0'
X       DC        PL4'-41503.22'
Y       DC        PL3'2011.4'
CZERO   DC        PL1'0'
C5      DC        PL1'5'
Z       DS        CL4
R       DS        CL3
        END       PLACE
```

7.5
THE COMPARE DECIMAL OPERATION

This SS instruction (mnemonic code CP) provides the mechanism for comparing the values in two packed-decimal fields, so that a subsequent branch can be set up. The general form is

name CP *operand1,operand2*

where *operand1* and *operand2* may each be up to 16 bytes long. If the operands are not of equal length, the shorter one is automatically padded with the appropriate number of high-order zeros prior to the comparison.

As a result of a CP execution, the condition code is set as follows:

Condition Code	*Table Value*	*Set When*
00	8	*Operand1* = *operand2*
01	4	*Operand1* < *operand2*
10	2	*Operand1* > *operand2*

EXAMPLE 28

One hundred four-byte packed-decimal fields (some of which may be negative) are stored consecutively in memory, the address of the first being ARRAY. Write a program to find the field whose value is the greatest and store it at T.

Solution Procedure

1. Set T = −9999999. This value was chosen because it is the smallest that can be stored in a four-byte packed-decimal field.
2. Load the constant 100 into a general register for use in counting iterations.
3. Compare T with the next member of the array (in the first iteration T is

compared with ARRAY). Whenever a number larger than T is found, substitute it for T; otherwise, do not alter T.

4. If the last member of the array has been examined, terminate the program; the largest element is in T. If the last member has not yet been examined, go back to the preceding step.

Solution Program

```
*Name    Operation  Operand                 Comment
         START      5000
BIG      BALR       15,0
         USING      *,15
         ZAP        T,SMALL
         L          10,COUNT
LOOP     CP         T,ARRAY
         BC         10,MODIFY               BYPASS REPLACEMENT
                                            IF T > THIS FIELD
         MVC        MODIFY-2(2),LOOP+4      CHANGE ADDRESS IN
                                            SECOND MVC
         MVC        T,ARRAY                 REPLACE T WITH NEW
                                            VALUE
MODIFY   LH         11,LOOP+4
         A          11,FOUR                 ADD 4 TO 2ND OPER-
                                            AND OF CP INSTRUC-
                                            TION
         STH        11,LOOP+4
         BCT        10,LOOP
         EOJ
SMALL    DC         XL4'9999999D'
ARRAY    DS         100CL4
T        DS         CL4
         DS         0F
COUNT    DC         F'100'
FOUR     DC         F'4'
         END        BIG
```

The flowchart for this program is given in Figure 7.1. In addition to the use illustrated here the procedure represents a general scheme for locating the smallest element in an array, the elements with largest and smallest absolute values, the element nearest in value to that of a specified constant, and so forth. In addition, this example contains the nucleus of one method of sorting in which the largest element is located, then the largest of the remaining elements, and so forth.

Discussion

The first instruction in the iterative loop compares the current value in T with an element of the array. Whenever the value of T is equal to or greater than the element of the array being examined, the branch to MODIFY is taken so that the second operand address of CP (located at LOOP+4) can be incremented by 4 and the next comparison made. Whenever an element greater than T is found, however, the pair of MVC instructions places it in T before the statement named LOOP is

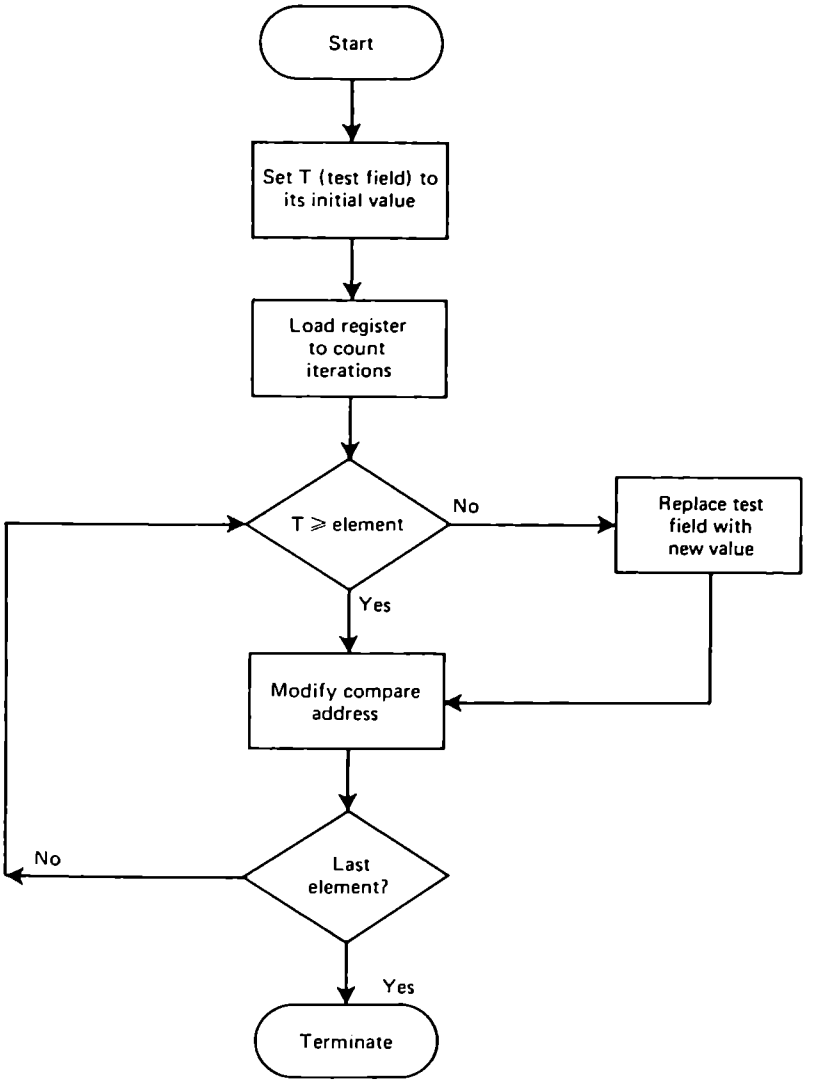


Figure 7.1 Flow chart for Example 28

modified. The first MVC obtains the address of the element from the CP instruction and moves it to the second operand address of the second MVC. (This is necessary because the address of the element to be moved is not known in advance.) Then the second MVC is able to move the element to T.

7.6 THE TRANSLATE INSTRUCTIONS

System/360 Assembler Language greatly facilitates character-matching operations by providing the translate and test (TRT) and translate (TR) instructions. These very powerful operations may be used to test data or results against any or all of the 256 EBCDIC characters automatically.

7.6.1 The TRT Instruction

This SS instruction operates on single bytes from left to right and has the general form

name TRT *operand*1,*operand*2

where *operand*1 refers to the first of one or more bytes and *operand*2 is the address of the first byte in a table whose maximum (and most common) length is 256 bytes. When this instruction is executed, each byte in *operand*1, starting with the leftmost byte, is treated as an eight-bit binary integer. The value represented by that integer is added to the address of *operand*2's first byte. This gives an address of some other byte in *operand*2's table. The byte in *that* address is compared to the one currently under scrutiny from *operand*1. Subsequent action based on that comparison is as follows:

1. If *operand*2's byte is zero, the process is repeated using the next sequential byte for *operand*1.
2. If *operand*2's byte is not zero or if there are no more bytes to be examined in *operand*1, the operation terminates, with the condition code set as follows:

Condition Code	*Table Value*	*Set When*
00	8	No match between bytes
01	4	Match found, more of *operand*1 remains
10	2	Match found, all of *operand*1 tested

3. If a match has been found between a byte in *operand*1 and a byte referred to in *operand*2, the address of *operand*1's byte is automatically stored in bits 8-31 of General Register 1 and the byte against which it was matched is placed in bits 24-31 of General Register 2.

For example, the EBCDIC bit configuration for a G is 11000111 (Table 2.1) which works out to a hexadecimal value of C7 (or 199_{10}). Thus if a particular byte had a G in it and were being tested by a TRT instruction against a table beginning at LOOK, it would test that G against the contents of LOOK + 199. A general flow chart for the TRT operation is given in Figure 7.2 and the more common EBCDIC representations, expressed as hexadecimal values, are shown in Table 7.3.

EXAMPLE 29

The address of the first of 100 consecutive bytes is STRING. Write a program to count the number of occurrences of the letter S in this string.

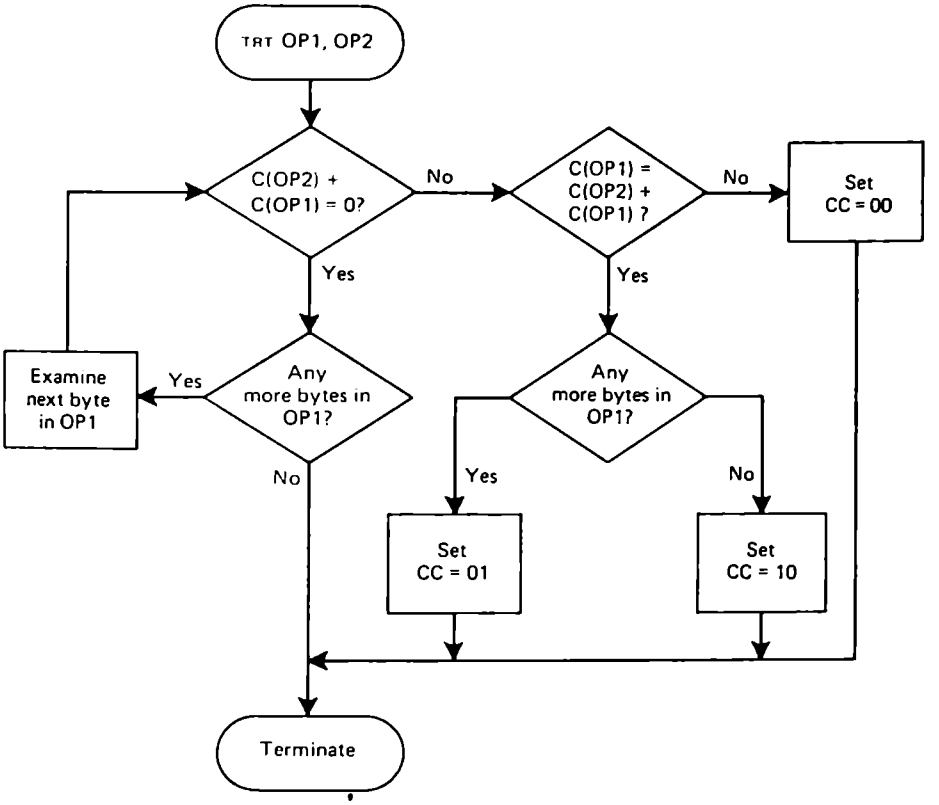


Figure 7.2 Action of the TRT instruction (C is read as "contents of"; OP1 refers to the leftmost byte of the first operand; OP2 refers to the leftmost byte of the second operand; C(OP1) refers to the contents of the byte of the first operand currently being tested.)

TABLE 7.3. EBCDIC CHARACTER CODES EXPRESSED WITH HEXADECIMAL DIGITS

Character	EBCDIC Code	Character	EBCDIC Code	Character	EBCDIC Code
A	C1	Q	D8	6	F6
B	C2	R	D9	7	F7
C	C3	S	E2	8	F8
D	C4	T	E3	9	F9
E	C5	U	E4	.	4B
F	C6	V	E5	,	6B
G	C7	W	E6	=	7E
H	C8	X	E7	(	4D
I	C9	Y	E8	)	5D
J	D1	Z	E9	+	4E
K	D2	0	F0	–	6D
L	D3	1	F1	<	4C
M	D4	2	F2	>	6E
N	D5	3	F3	'	7D
O	D6	4	F4	*	5C
P	D7	5	F5	$	5B

Solution Procedure

1. Initialize a field to be used as a counter.
2. Generate a table of 256 consecutive bytes which are all zero except the 227th. This is the position, discussed below, corresponding to the letter S.
3. Use Translate and Test to locate the next S.
4. If there are no more occurrences of the letter S, terminate the program; if an S occurs as the last byte of STRING, add 1 to the counter and terminate; otherwise, add 1 to the counter, delete the S that was located and go back to the previous step.

Solution Program

```
Name      Operation  Operand              Comment
          START      5000
LOCATE    BALR       15,0
          USING      *,15
          SP         CTR,CTR
          MVC        LIST+226(1),X'1C'    A NONZERO IN 227TH
                                          BYTE
LOOP      TRT        STRING(100),LIST
          BC         8,OVER               NO S FOUND
          BC         2,ADD
          AP         CTR,X'1C'            ANOTHER S FOUND
          MVI        0(1),X'00'           DELETE THE S JUST FOUND
          BC         15,LOOP
ADD       AP         CTR,X'1C'            ENTIRE STRING PROCESSED
OVER      EOJ
CTR       DS         CL2
STRING    DS         100C
LIST      DC         256X'00'
          END        LOCATE
```

Discussion

This example could have been programmed very simply by a loop that tested each byte of STRING and incremented a counter whenever one was located. This method would have required 100 iterations, however; one for each byte tested. The method used, in comparison, requires as many iterations as the number of occurrences of the letter S or, at most, one more.

In the example, the first operand, STRING, is defined as a string of 100 bytes; the second operand, LIST, is defined as a set of 256 bytes, all of which are zero except the 227th, which is +1 (the definition of LIST with DC is modified as a result of the MVC instruction). The reason the 227th byte is made nonzero is that occurrences of the letter S are being sought, and the bit configuration for S is 11100010 which, interpreted as a binary integer, is 226 (base 10). Adding 226 to the second operand address produces the address of the 227th byte in the table. If others characters were being sought at the same time, their bit configurations, interpreted as binary integers, would be used to locate other nonzero bytes in the list.

Notice that the first operand address of the MVI instruction in the Solution Program is 0(1). This is the explicit specification of a displacement of 0 modified by positions 8–31 of General Register 1 used as a base register. Because TRT places the address of a byte in General Register 1, this is a convenient method for writing an instruction with a variable address without, at the same time, having to modify such address. The second operand of the MVI instruction may have any bit configuration except that of the letter S in this example; subsequent executions of TRT will thereby pass it by.

7.6.2 The TR Instruction

The TR instruction is somewhat similar to TRT in that the second operand is a table and a particular entry in it is found by using a byte in the first operand, converting it to its binary equivalent and adding that many bytes to the first entry in the table. However, once the proper entry is found, it replaces the byte in the first operand. For example, let us assume that TRY is a two-byte field containing the characters * and $, and PILE is the address of the first byte of a 256-byte table. Then the statement

```
TR      TRY(2),PILE(256)
```

will cause the following: Since * is stored as $5C_{16}$ (see Table 7.3) or 92_{10}, the * in TRY will be replaced by the contents of PILE+92. Then, the $ in TRY+1 will be replaced by the character in PILE+91 (since a $ is stored as $5B_{16}$).

EXAMPLE 30

A string of 175 bytes contains numeric, alphabetic, and special characters; the address of the first byte is STRING. Write a program to set each of the alphabetic and special characters to 0 and to replace each of the numeric characters with its tens complement (the difference between itself and 10).

Solution Procedure

Since EBCDIC stores the ten numeric characters as F0 through F9 (240_{10} through 249_{10}) we can set up a 256-byte table (which we shall call LIST) in which all the entries are zeros except LIST+240 through LIST+249. These will be set at 10 through 1, respectively. Then the whole task can be done with a single TR execution.

Solution Program

```
Name      Operation   Operand          Comment
          START       6000
CHANGE    BALR        15,0
          USING       *,15
          TR          STRING,LIST
          EOJ
STRING    DS          175C
```

```
Name    Operation  Operand                          Comment
LIST    DC         15XL16'0'                        0 IN LIST THROUGH
                                                    LIST+239
        DC         XL10'FAF9F8F7F6F5F4F3F2F1'       10's COMPLEMENTS IN
                                                    LIST+240-LIST+249
        DC         XL6'0'                           0 IN LIST+250-
                                                    LIST+255
        END        CHANGE
```

7.7 EDITING INSTRUCTIONS

The results of packed-decimal operations are in no shape for convenient human scrutiny. Before they can be easily read and interpreted, their form must be altered and their contents supplemented by various symbols. Much of this can be accomplished by means of the System/360's editing instructions.

7.7.1 The Edit Instruction

The basic mechanism for inserting the various symbols in a numeric field consists of setting up an *editing mask* in which each position of the field to be displayed is described in detail. This mask is set up in a storage area and the output value is superimposed on that field, whereupon the required insertions are made in accordance with the directions implied in the mask. This superposition is achieved via the Edit instruction, whose basic form is

name ED *mask,output*

where *mask* refers to an address containing the editing mask and *output* represents the source from which the data are to be transferred for superposition. Since the Edit operation destroys the mask, it is usually stored someplace and placed in the working area each time it is needed by means of an MVC. As many as 256 bytes may be handled by a single Edit instruction, with processing going from left to right. The data to be edited (superimposed onto the mask) must be in packed-decimal form.

The edit mask is constructed as follows: Each numeric digit (four bits) in the data field is represented by a character (a full byte) in the mask called a *digit selector.* We shall show this selector as D for explanatory purposes, but let us bear in mind that when we set up an actual mask field, we shall use the hexadecimal value x'20' for that purpose.

The string of digit selectors is usually preceded by a *fill character* which automatically takes care of zero suppression. That is, the specification of a fill character (referred to here as F) will set up a mechanism whereby the

numerical value is scanned and all high-order zeros are replaced with fill characters.

One of several types of fill characters can be used by specifying the appropriate representation as part of the edit mask. Although the blank (which we shall denote by ♭ but whose hexadecimal equivalent is 40) is the most common fill character, it is by no means the only one used. Any character can be specified as a fill character by designating the hexadecimal equivalent of its internal EBCDIC code.

Suppose, for example, that we wished to use the asterisk as a fill character for a seven-digit packed-decimal field named X. The asterisk's internal code is 5C. Hence, our edit mask (labelled EDMSK) will consist of eight bytes—an asterisk followed by seven digit selectors. If our output field (also eight bytes long) is called OUT, the allocation statements would read

```
OUT     DS   CL9
EDMSK   DC   X'5C2020202020202020'
X       DS   PL4
```

and the editing sequence would be

```
MVC   OUT,EDMSK
ED    OUT,X
```

Under these circumstances, the values of X shown in the left-hand column below will produce the OUT values shown on the right after being edited by EDMSK:

X	OUT
42 06 98 1C	4206981
00 24 06 2C	**24062
01 10 49 0D	*110490

Symbols like $, =, or – would work the same way. Note that the sign is not included. This is fine as long as the programmer is sure that the field to be edited will never contain a negative value. When this is not the case, a value of $(60)_{16}$(a minus sign) is specified in the edit mask at the position where the sign is to appear. When this specification is included, the sign will appear only if the value is negative.

The placement of a decimal point or comma in an edited field is controlled by specifying the respective internal representation at the desired position in the mask. This is 6B for a comma and 4B for a period. If zero suppression is called for by specifying a fill character and if there is a zero immediately to the right of where a decimal point is to appear, that point will not stop the suppression process. In order to restrict zero suppression to all places

TABLE 7.4 EFFECT OF THE ED INSTRUCTION USING THE MASK

X'4020206B2020214B2020404060'
F D D , D D D ' . D D b b -

Unedited Number	Edited Number
3 2 9 4 5 6 8 C	32,945.68bbb
0 2 7 5 4 1 0 D	b2,754.10bb-
0 0 1 0 7 5 6 C	bbb107.56bbb
0 0 0 6 0 4 2 D	bbbb60.42bb-
0 0 0 0 1 7 9 C	bbbbb1.79bbb
0 0 0 0 0 2 8 D	bbbbb.28bb-
0 0 0 0 0 0 4 C	bbbbbb.04bbb
0 0 0 0 0 0 0 C	bbbbbb.00bbb

to the left of the decimal point, a $(21)_{16}$ (which we shall call D') is specified in the editing mask at the position immediately to the left of the decimal point. Table 7.4 shows the effect of the ED instruction on several packed-decimal values for a particular mask.

When an ED instruction is executed, the condition code is set as follows:

Condition Code	*Table Value*	*Set When*
00	8	Number being edited = 0
01	4	Number being edited < 0
10	2	Number being edited > 0

The ED instruction will operate according to ASCII if bit 12 of the PSW is 1.

EXAMPLE 31

PDF is a packed-decimal field four bytes long. This field is to be prepared for output. It is to be converted from packed-decimal to zoned-decimal format, to have a decimal point inserted between the second and third digits from the right, to have its leading zeros (if any) suppressed, and to have a comma inserted between the hundreds' place and thousands' place if there is a significant digit to the left.

Solution Procedure

1. Move the editing pattern to the result location.
2. Edit the packed-decimal field into the result location.

Solution Program

```
Name    Operation   Operand
        START       10000
EDIT    BALR        15,0
        USING       *,15
        MVC         ZDF,PATTRN
        ED          ZDF,PDF
        EOJ
PDF     DS          CL4
ZDF     DS          CL12
PATTRN  DC          X'4020206B2020214B20204060'
        END         EDIT
```

7.7.2 The Edit and Mark (EDMK) Instruction

This SS instruction operates like ED with an additional facility. When zero suppression is called for, it stores the address of the first significant (nonzero) digit in General Register 1. By being able to refer to this address, the programmer can place a desired character to the left of that point. Since that position may vary for a given field, depending on its current value, the programmer has, in effect, a way of providing a floating symbol.

For example, if we wanted to set up a value for eventual output in which a $ is to appear immediately to the left of the first significant digit, we would set up an edit mask as before and move the value into the edit field with EDMK rather than ED. Since the address of the first significant digit will now be in Register 1, we can subtract one from that address by the instruction BCTR 1,0 and move our $ into the new address specified by Register 1. Specifically, if X is a field containing

0	0	2	1	3	4	0	D

and we define

```
EDMSK   DC      X'4020206B2020214B202060'
OUTFLD  DS      CL11
```

then the sequence

```
        MVC     OUTFLD(11),EDMSK
        EDMK    OUTFLD(11),X
        BCTR    1,0
        MVI     0(1),C'$'
```

will produce an OUTFLD value of

b	b	$	b	2	1	3	.	4	0	–

Had X's value been

0	4	0	1	6	0	9	C

the OUTFLD value would have been

b	$	4	.	0	1	6	.	0	9	b

EXAMPLE 32

Program Example 31 once more, using the same definitions of PDF, PATTRN, and ZDF, but this time placing the minus sign, if the number is negative, in the byte to the left of the first significant digit or the decimal point, whichever comes first.

Solution Procedure

1. Place the address of the decimal point in ZDF in General Register 1.
2. Edit PDF using the Edit and Mark instruction.
3. Reduce the address stored in General Register 1 by 1.
4. Move the contents of the sign position to the byte whose address is in General Register 1.
5. Move the fill character to the original location of the sign.

Solution Program

```
Name     Operation   Operand
         START       8000
MARK     BALR        15,0
         USING       *,15
         L           1,ZDF7
         MVC         ZDF,PATTRN
         EDMK        ZDF,PDF
         BCTR        1,0
         MVC         0(1,1),ZDF+11
         MVC         ZDF+11(1),ZDF
         EOJ
         DS          0F
ZDF7     DC          A(ZDF+7)
PATTRN   DC          X'4020206B2020214B2020'
ZDF      DS          CL12
PDF      DS          CL4
         END         MARK
```

Discussion

In this program we have provided for the contingency in which there may be no significant digits ahead of the decimal point. This is done by storing the *address* of the decimal point itself) in ZDF7 and placing that address in Register 1. Then if there are no significant digits we can place our minus sign (if we need one) accordingly. If there are significant digits, EDMK will change the address in Register

l and proper placement of the sign will still be assured. Since the A code in a DC statement requires alignment at a word boundary (see Table 4.6), we have preceded the DC with the appropriate DS.

PROBLEMS

1. In which formats may decimal numbers appear in System/360, and how is conversion from one to the other accomplished?
2. Which of the formats for decimal numbers is used during computation?
3. What are the major differences between arithmetic as carried out with the decimal instructions and as carried out by the binary instructions?
4. What factors would tend to cause a problem programmer to use the decimal arithmetic instructions rather than either of the other types available in System/360?
5. Why do all the decimal arithmetic instructions use the SS format?

Write programs to accomplish the following:

6. Compute the sum of +123, +42, and −110, where these quantities are stored in packed-decimal format, and store the sum in a 3-byte field.
7. Compute the sum of +12.3, +4.2, and −11.0, where these quantities are stored in packed-decimal format, and store the sum in a 3-byte field.
8. Compute the sum of 50 packed-decimal numbers, each of which is stored in a 5-byte field, and store it in a 6-byte field.
9. The centigrade temperature, C, is given. Compute the equivalent Fahrenheit temperature, F, according to the formula: F = 32 + 9/5 C. Consider that both C and F are expressed in the packed-decimal format.
10. The Fahrenheit temperature, F, is given. Compute the equivalent centigrade temperature, C, if F is expressed in zoned-decimal format and C is converted to zoned-decimal format after computation.
11. An array of packed-decimal integers is defined as follows:

Name	*Operation*	*Operand*
ARRAY	DS	151CL4

    Compute the arithmetic mean of this array, rounding the result to the nearest tenth and storing the result in MEAN.
12. For the array of Exercise 6, find the median value by sorting the numbers into descending order, locating the 76th, and storing it in MEDIAN.
13. For the array of Problem 6, and after sorting as in Problem 7, compute the value of the first quartile by interpolating between the 37th and 38th numbers, rounding to the nearest hundredth, and storing the result in QTILE.

14. Considering the array of Problem 6 as a string of 604 bytes, find where the first digit 5 occurs and store its address in FIVE.
15. Edit the result of Problem 6 for output by converting it to zoned-decimal format, suppressing leading zeros, inserting a decimal point, and inserting commas as required between triplets of digits to the left of the decimal point. If the magnitude of the mean is less than one, express the result with a zero to the left of the decimal point.

Chapter 8

Floating-Point Arithmetic

Until now, relatively little attention has been paid to numbers other than integers. This is because the instructions in System/360 for both binary and decimal arithmetic treat numbers only as integers. The responsibility for keeping track of the decimal or binary point, when numbers with fractional parts are processed, rests entirely with the programmer. This becomes a time-consuming burden for problem programs in which the computation is lengthy and in which the range of the possible values of numerous variables is wide.

The purpose of having instructions for performing arithmetic in the floating-point mode is to relieve the programmer of the necessity of paying attention to radix-point placement in all but unusual circumstances. The floating-point arithmetic instructions operate under the assumptions that the operands are in either the 32-bit (*short operand*) or 64-bit (*long operand*) floating-point format and that, when reference is made to registers, it is the floating-point registers (numbered 0, 2, 4, and 6) that are meant. Otherwise, these instructions are very similar to the binary-arithmetic instructions from the programmer's point of view.

Reviewing the hexadecimal data formats briefly, both long and short formats use bit position 0 for the sign (0 for plus and 1 for minus), and both use bit positions 1-7 for the characteristic. The short operand format

uses positions 8–31 for the fraction, and the long operand format uses positions 8–63 for this purpose. In both cases the assumed position of the hexadecimal point is between bit positions 7 and 8.

The characteristic is interpreted as a 7-bit positive integer greater by 64 than the exponent. The exponent, in turn, is interpreted as the power of 16 by which the fraction is multiplied to obtain the true value. Either the exponent or the difference between the characteristic and 64, therefore, is the number of digits the hexadecimal point must be moved from its assumed position to represent this true value. Positive exponents (or differences) are associated with movement to the right, and negative exponents (or differences) are associated with movement to the left.

The fraction consists of either six (for short operands) or fourteen (for long operands) binary-coded hexadecimal digits, four bits being used to represent each hexadecimal digit. Floating point fields are ordinarily used in normalized form: that is, the fraction has its leftmost significant digit in bit-positions 8–11. The student may refresh his memory further by consulting Sec. 2.3.4.

Since these two operand formats are available, System/360 presents duplicate repertoires of floating-point instructions. Those dealing with short operand formats contain an E in the mnemonic operation code; a D is used to denote a long operand floating-point instruction. Further replication is supplied by providing RX and RR versions for almost all of the floating point instructions.

8.1 LOAD, STORE, ADD, DIVIDE, AND COMPARE INSTRUCTIONS

In general, the floating-point instructions provide a functional parallel to the fixed point set. Consequently, it will usually not be necessary to explain their operation in detail.

8.1.1 Load Instructions

There are twelve types of floating-point Load instructions. We shall concern ourselves at this point with the four basic ones.

A floating-point value is brought into a floating-point register with an instruction beginning with L. The operation code is LE if the value is a short operand coming from memory and LER if the short operand is coming from another floating-point register. Similarly, the LD and LDR handle long operands. Thus if *freg* refers to a particular floating-point register to be used as a destination, *freg'* represents a floating-point register to be used as a source, *op* is a short operand in memory and *opop* is a long operand, we can summarize these four instructions as

```
LD      freg, opop
LE      freg, op
LDR     freg, freg'
LER     freg, freg'
```

The condition code is not set.

8.1.2 Store Instructions

Duplication of data from a floating-point register into an area in storage is handled by the STE or STD instruction, depending on the length of the destination operand. Thus

```
STE     freg'.op
```

stores the high-order (leftmost) 32 bits of the designated floating-point register in *op*. The instruction

```
STD     freg',opop
```

causes all 64 bits in *freg'* to be duplicated in *opop*.

8.1.3 Floating-Point Addition

Eight instructions are provided for floating-point addition. In all cases the first operand specifies a floating-point register which is to contain the resulting sum. When the "E" form is used, the sum is stored in the leftmost 32 bits of the destination register, with the right half remaining unchanged.

Two types of floating-point addition are performed. In *normalized* addition (ADR, AD, AER, AE) the exponent is automatically adjusted so that the fractional portion does not begin with a zero. When *unnormalized* addition is specified (by AWR, AW, AUR, or AU), this adjustment is not made and the fractional portion may contain leading zeros. General forms are shown herewith:

```
Normalized      { AD    freg,op
                { ADR   freg,freg'
                { AE    freg,opop
                { AER   freg,freg'

Unnormalized    { AU    freg,op
                { AUR   freg,freg'
                { AW    freg,opop
                { AWR   freg,freg'
```

As a result of floating-point addition, the condition code is set as follows:

Condition Code	*Table Value*	*Set When*
00	8	Fraction portion of sum = 0
01	4	Fraction portion of sum < 0
10	2	Fraction portion of sum > 0
11	1	Exponent portion of sum overflows

8.1.4 Floating-Point Division

Four division instructions are provided. The DE and DER instructions produce a quotient having a length of six hexadecimal digits, using the leftmost half of the destination register.

The quotient length for the DD and DDR instructions is 14 digits. Remainders are not preserved in any case and all quotients are normalized. General forms are as follows:

```
DE      freg,op
DER     freg,freg'
DD      freg,opop
DDR     freg,freg'
```

Two RR instructions are provided for an operation which can be thought of as a specific type of division.

```
HER     freg,freg'
HDR     freg,freg'
```

duplicate the contents (half or entire) of *freg'* in *freg*, and shift the value one bit to the right, thus halving it. Normalization follows as required.

8.1.5 Comparison

The four floating-point comparison instructions are

```
CE      freg,op
CER     freg,freg'
CD      freg,opop
CDR     freg,freg'
```

In all cases, the two operands are compared algebraically and the condition code is set as follows:

Condition Code	*Table Value*	*Set When*
00	8	*freg* = second operand
01	4	*freg* < second operand
10	2	*freg* > second operand

EXAMPLE 33

N is a floating-point quantity whose value is 309.19. Compute its square root, and store it in ROOT. Write the program twice, once where N is in short-operand format and once where N is in long-operand format.

Solution Procedure

1. Load N into a floating-point register.
2. Load a first approximation to the square root into a floating-point register. In specific problem programs, knowledge of the data being processed may be used to choose a first approximation effectively and, thereby, to reduce the number of iterations required to achieve any desired degree of accuracy. In this general example, however, N is used as the first approximation to its own square root.
3. Compute an improved approximation using Newton's formula:

$$X_{i+1} = .5\left(\frac{N}{X_i} + X_i\right)$$

   In this formula, X_i is an approximation to the square root of N and X_{i+1} is the following approximation. By repeated application of this formula, square root can be computed to any desired degree of accuracy. (Most roots are irrational numbers and cannot be expressed exactly by a finite number of digits.)
4. Test the improved approximation. If it is sufficiently accurate, store it in ROOT; if it is not, return to the previous step. There are several ways to test an approximation. In this example an approximation is acceptable if it is exactly equal to the previous approximation. (Because this test can never be met when the last bit of the fraction oscillates between 0 and 1, it is not a general test and is used only to avoid introducing too many new instructions in a single example.)

Solution Flowchart

The logic of this example is presented graphically in Figure 8.1. Note that when more realistic termination tests are used, the flowchart retains its form with only the contents of the diamond being altered.

Solution Programs

Name	*Operation*	*Operand*
	START	8000
NEWTON	BALR	15,0
	USING	*,15
	LE	2,N
	LER	6,2
LOOP	LER	4,6
	LER	6,2
	DER	6,4

Name	*Operation*	*Operand*
	START	8000
NEWTON	BALR	15,0
	USING	*,15
	LD	2,N
	LDR	6,2
LOOP	LDR	4,6
	LDR	6,2
	DDR	6,4

Name	*Operation*	*Operand*
	AER	6,4
	HER	6,6
	CER	6,4
	BC	7,LOOP
	STE	6,ROOT
	EOJ	
	DS	0F
N	DC	E'309.19'
ROOT	DS	F
	END	NEWTON

Name	*Operation*	*Operand*
	ADR	6,4
	HDR	6,6
	CDR	6,4
	BC	7,LOOP
	STD	6,ROOT
	EOJ	
	DS	0D
N	DC	D'309.19'
ROOT	DS	D
	END	NEWTON

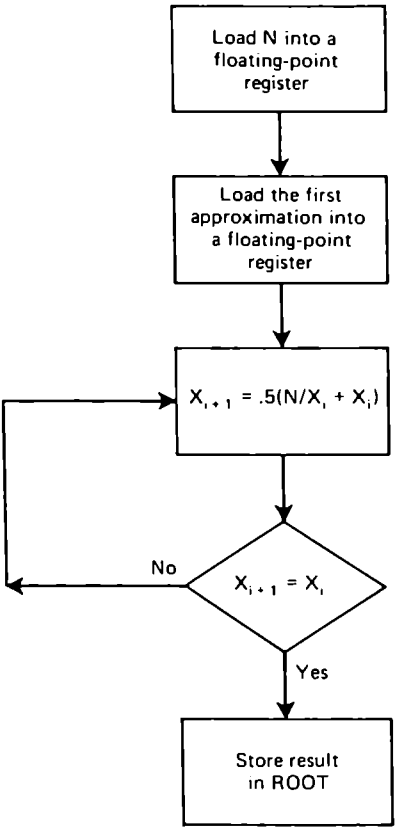

Load N into a floating-point register
Load the first approximation into a floating-point register
$X_{i+1} = .5(N/X_i + X_i)$
No
$X_{i+1} = X_i$
Yes
Store result in ROOT

Figure 8.1 Flow chart for Example 33

8.2 THE LOAD POSITIVE AND SUBTRACT INSTRUCTIONS

The choice of instructions being introduced in this section is arbitrary, as was the case before. This selection happens to be appropriate for the next example.

8.2.1 The LPER and LPDR Instructions

These RR instructions place the contents of the second operand, with sign made positive (the absolute value) in the first operand. The condition code is set as follows:

Condition Code	*Table Value*	*Set When*
00	8	Result = 0
10	2	Result > 0

Others are not used. Thus, if Floating-Point Register 2 contains the equivalent of −41 and we say

```
LPER   6,2
```

Floating-Point Register 6 will contain +41 and the condition code will be set to 10.

8.2.2 Floating-Point Subtraction

The eight floating-point subtraction instructions parallel those for addition (see Sec. 8.1.3). In all cases, the second operand is subtracted from the first, with the resulting difference stored in the first operand. Using our previous operand nomenclature, the general forms are as follows:

Normalized	SD	*freg,op*
	SDR	*freg,freg'*
	SE	*freg,opop*
	SER	*freg,freg'*
Unnormalized	SU	*freg,op*
	SUR	*freg,freg'*
	SW	*freg,opop*
	SWR	*freg,freg'*

For all floating-point subtractions, the condition code is set as follows:

Condition Code	*Table Value*	*Set When*
00	8	Difference = 0
01	4	Difference < 0
10	2	Difference > 0
11	1	Overflow in exponent portion of *freg*

EXAMPLE 34

Repeat Example 33 with the following change: terminate the computation when the difference between the new and old approximations is less than or equal to .001.

Solution Procedure

In the previous example, the termination test, shown in the diamond of Figure 8.1, was $X_{i-1} = X_i$; that is, the computation was to terminate whenever these quantities became equal. In this example the termination test is as follows:

$$|X_{i-1} - X_i| \leq .001$$

It is necessary to use the absolute value of the difference rather than just the algebraic difference because the latter may be either positive or negative. An alternate approach, however, is to utilize the algebraic difference and to make the termination test a combination of the following two criteria:

$$X_{i+1} - X_i \leq .001$$
$$X_{i+1} - X_i \geq -.001$$

This combination test is equivalent to first determining the sign of $X_{i+1} - X_i$ and then testing whether a positive difference is less than or equal to .001 or a negative difference is greater than or equal to $-.001$.

Solution Programs

Name	*Operation*	*Operand*	*Name*	*Operation*	*Operand*
	START	8000		START	8000
NEWTON	BALR	15,0	NEWTON	BALR	15,0
	USING	*,15		USING	*,15
	LE	2,N		LD	2,N
	LER	6,2		LDR	6,2
LOOP	LER	4,6	LOOP	LDR	4,6
	LER	6,2		LDR	6,2
	DER	6,4		DDR	6,4
	AER	6,4		ADR	6,4
	HER	6,6		HDR	6,6
	SER	4,6		SDR	4,6
	LPER	4,4		LPDR	4,4
	SE	4,TEST		SD	4,TEST
	BC	3,LOOP		BC	3,LOOP
	STE	6,ROOT		STD	6,ROOT
	EOJ			EOJ	
	DS	0F		DS	0D
N	DC	E'309.19'	N	DC	D'309.19'
TEST	DC	E'.001'	TEST	DC	D'.001'
ROOT	DS	F	ROOT	DS	D
	END	NEWTON		END	NEWTON

8.3 FLOATING-POINT MULTIPLICATION

The four floating-point multiplication instructions are similar in structure to those for division. In all cases, the product is normalized and stored in the first operand. Thus

ME	*freg,op*
MD	*freg,opop*
MER	*freg,freg'*
MDR	*freg,freg'*

The condition code is not affected.

EXAMPLE 35

Repeat Example 33 with the following changes: (1) Let the first approximation to the square root of N be N/2; (2) use Bailey's method of computing improved approximations; and (3) terminate the computation when $|N - X_{i+1}^2| < .001$.

Solution Procedure

The basic approach used in the two previous examples is also used here. Bailey's formula for improving approximations to a square root is as follows:

$$X_{i+1} = X_i \frac{X_i^2 + 3N}{3X_i^2 + N}$$

The formula can be generalized for use in extracting any integral root of a number. If n designates the root ($n = 2$ for square root; $n = 3$ for cube root, and so forth), the formula becomes

$$X_{i+1} = X_i \frac{(n-1)X_i^n + (n+1)N}{(n+1)X_i^n + (n-1)N}$$

The problem programmer is often faced with the task of writing a program that is in some sense most efficient, efficiency being measured by speed, storage required, accuracy, and/or elapsed time until the program is running. Choice of algorithms can make a significant difference. Bailey's method requires, for example, more computation per iteration than Newton's, but converges to a solution faster and needs fewer iterations.

Solution Programs

Name	*Operation*	*Operand*
	START	8000
BAILEY	BALR	15,0
	USING	*,15
	LE	2,N
	HER	6,2
LOOP	LER	4,6
	MER	4,4
	LER	0,2
	AER	0,2
	AER	0,2
	AER	0,4
	MER	6,0
	ME	4,THREE

Name	*Operation*	*Operand*
	START	8000
BAILEY	BALR	15,0
	USING	*,15
	LD	2,N
	HDR	6,2
LOOP	LDR	4,6
	MDR	4,4
	LDR	0,2
	ADR	0,2
	ADR	0,2
	ADR	0,4
	MDR	6,0
	MD	4,THREE

Name	*Operation*	*Operand*	*Name*	*Operation*	*Operand*
	AER	4,2		ADR	4,2
	DER	6,4		DDR	6,4
	LER	4,6		LDR	4,6
	MER	4,4		MDR	4,4
	SER	4,2		SDR	4,2
	LPER	4,4		LPDR	4,4
	SE	4,TEST		SD	4,TEST
	BC	11,LOOP		BC	11,LOOP
	STE	6,ROOT		STD	6,ROOT
	EOJ			EOJ	
	DS	0F		DS	0D
N	DC	E'309.19'	N	DC	D'309.19'
ROOT	DS	F	ROOT	DS	D
THREE	DC	E'3'	THREE	DC	D'3'
TEST	DC	E'.001'	TEST	DC	D'.001'
	END	BAILEY		END	BAILEY

8.4 ADDITIONAL LOAD INSTRUCTIONS

The floating-point repertoire includes six additional instructions for shifting information between floating-point registers.

LNER and LNDR (Load Negative) instructions work like LPER and LPDR (see Sec. 8.2.1), except that the sign of the second operand value is set to negative instead of positive. The condition code is set to 00 if the fractional portion is zero; otherwise (if it is < 0) the code is set to 01.

The LCER and LCDR instructions (Load Complement) duplicate the second operand value in the first operand, but with the opposite sign. Condition code settings are as follows:

Condition Code	*Table Value*	*Set When*
00	8	Result = 0
01	4	Result $<$ 0
10	2	Result $>$ 0

The sign of a value in a floating-point register may be tested rather than changed by the LTER or LTDR instructions. When we say

```
        LTER    freg,freg'
```

or

```
        LTDR    freg,freg'
```

the contents of *freg'* are loaded into *freg* (*freg'* and *freg* can be the same) and the condition code is set as for LCER or LCDR.

EXAMPLE 36

Compute the roots of the quadratic equation, $AX^2 + BX + C = 0$, for both short-operand and long-operand values of A, B, and C. Store the real parts of the roots in R_1 and R_2 and the corresponding imaginary parts in I_1 and I_2.

Solution Procedure

1. Compute $D = B^2 - 4AC$.
2. Compute the square root of $|D|$.
3. If $D = 0$, set $R_1 = R_2 = -B/2A$ and $I_1 = I_2 = 0$. If $D > 0$, set $I_1 = I_2 = 0$ and set

$$R_1 = (-B + \sqrt{D})/(2A)$$

and

$$R_2 = (-B - \sqrt{D})/(2A).$$

If $D < 0$, set $R_1 = R_2 = -B/2A$ and set

$$I_1 = \sqrt{-D}/(2A)$$

and

$$I_2 = -\sqrt{-D}/(2A).$$

Use the method of Example 33 to compute the square root.

Solution Programs

Name	*Operation*	*Operand*
	START	5000
QUAD	BALR	15,0
	USING	*,15
	LE	0,B
	MER	0,0
	LE	4,FOUR
	ME	4,A
	ME	4,C
	SER	0,4
	LPER	3,0
	LER	6,2
LOOP	LER	4,6
	LER	6,2
	DER	6,4
	AER	6,4
	HER	6,6
	CER	6,4
	BC	7,LOOP
	DE	6,A
	HER	6,6
	LE	2,B
	LNER	2,2

Name	*Operation*	*Operand*
	START	5000
QUAD	BALR	15,0
	USING	*,15
	LD	0,B
	MDR	0,0
	LD	4,FOUR
	MD	4,A
	MD	4,C
	SDR	0,4
	LPDR	2,0
	LDR	6,2
LOOP	LDR	4,6
	LDR	6,2
	DDR	6,4
	ADR	6,4
	HDR	6,6
	CDR	6,4
	BC	7,LOOP
	DD	6,A
	HDR	6,6
	LD	2,B
	LNDR	2,2

Name	*Operation*	*Operand*
	DER	2,A
	HER	2,2
	SER	4,4
	LTER	0,0
	BC	4,CMPLEX
	STE	4,I1
	STE	4,I2
	BC	2,REALS
	STE	2,R1
	STE	2,R2
	BC	15,OVER
REALS	LER	4,2
	AER	2,6
	SER	4,6
	STE	2,R1
	STE	4,R1
	BC	15,OVER
CMPLEX	STE	2,R1
	STE	2,R2
	STE	6,I1
	LCER	6,6
	STE	6,I2
OVER	EOJ	
	DS	0F
A	DS	F
B	DS	F
C	DS	F
R1	DS	F
R2	DS	F
I1	DS	F
I2	DS	F
FOUR	DC	E'4'
	END	QUAD

Name	*Operation*	*Operand*
	DDR	2,A
	HDR	2,2
	SDR	4,4
	LTDR	0,0
	BC	4,CMPLEX
	STD	4,I1
	STD	4,I2
	BC	2,REALS
	STD	2,R1
	STD	2,R2
	BC	15,OVER
REALS	LDR	4,2
	ADR	2,6
	SDR	4,6
	STD	2,R1
	STD	4,R2
	BC	15,OVER
CMPLEX	STD	2,R1
	STD	2,R2
	STD	6,I1
	LCDR	6,6
	STD	6,I2
OVER	EOJ	
	DS	0D
A	DS	D
B	DS	D
C	DS	D
R1	DS	D
R2	DS	D
I1	DS	D
I2	DS	D
FOUR	DC	D'4'
	END	QUAD

Discussion

The solution program begins with the computation of $B^2 - 4AC$, which is left in Register 0 for later testing. The absolute value of $B^2 - 4AC$ is placed in Register 2, and its square root is extracted using the same program as that in Example 33. $\sqrt{|D|}/2A$ is computed and left in Register 6: $-B/2A$ is computed and left in Register 2; and Register 4 is set to 0.

$\sqrt{|D|}/(2A)$ and $-B/(2A)$ are computed prior to testing the sign of D to facilitate writing these instructions only once. If $D < 0$, the roots are complex and the branch to CMPLEX is taken. If $D \geq 0$, I1 and I2 are set to zero prior to the next BC. If $D > 0$, the roots are real and unequal, and the branch to REALS is taken; otherwise the roots are real and equal and both R1 and R2 are set equal to $-B/(2A)$.

At REALS, R1 is set to $-B/(2A) + \sqrt{D}/(2A)$ and R2 is set to $-B/(2A) -$

$\sqrt{D}/(2A)$. At CMPLEX, R1 and R2 are set to $-B/(2A)$; I1 is set to $\sqrt{-D}/(2A)$; and I2 is set to $-\sqrt{-D}/(2A)$.

EXAMPLE 37

Evaluate the polynomial $A + BX + CX^2 + DX^3$ using both short and long operand quantities for the coefficients and for the argument X. Let X range from 10.0 through 15.0 in steps of 0.1, and store the results in a table in memory.

Solution Procedure

Evaluation of polynomials greater than the first degree is carried out most efficiently, in terms of both speed and memory space, by first factoring them in a special manner. This manner of factoring is shown below for polynomials up to the fifth degree.

Degree	*Polynomial*	*Factored Form*
1	$A + BX$	A + BX
2	$A + BX + CX^2$	A + X(B + X(C))
3	$A + BX + CX^2 + DX^3$	A + X(B + X(C + X(D)))
4	$A + BX + CX^2 + DX^3 + EX^4$	A + X(B + X(C + X(D + X(E))))
5	$A + BX + CX^2 + DX^3 + EX^4 + FX^5$	A + X(B + X(C + X(D + X(E + X(F)))))

Comparing term by term evaluation with evaluation of the factored form, note that the former requires more multiplications, the difference becoming larger as the degree increases.

1. Load the starting value, final value, and increment of the argument X into three floating-point registers.
2. Evaluate the polynomial in the factored form

   $$A + X(B + X(C + X(D)))$$

   Computation begins in the innermost parentheses and proceeds outward in a series of multiplications and additions.
3. Store the result.
4. Compare the contents of the registers in which the initial and final values of X were placed. If they are unequal, add the increment to the former, modify the table storage instruction, and return to the evaluation step. If they are equal, the program is complete and is terminated.

Solution Program

Name	*Operation*	*Operand*
	START	8000
POLY	BALR	15,0
	USING	*,15
	LE	2,FIRST
	LE	4,LAST

Name	*Operation*	*Operand*
	START	8000
POLY	BALR	15,0
	USING	*,15
	LD	2,FIRST
	LD	4,LAST

Name	*Operation*	*Operand*
	LE	6,STEP
LOOP	LE	0,D
	MER	0,2
	SE	0,C
	MER	0,2
	AE	0,B
	MER	0,2
	AE	0,A
STORE	STE	0,TABLE
	CER	2,4
	BC	8,OVER
	AER	2,6
	LH	10,STORE+2
	AH	10,INCR
	STH	10,STORE+2
	BC	15,LOOP
OVER	EOJ	
	DS	0F
A	DS	F
B	DS	F
C	DS	F
D	DS	F
FIRST	DC	E'10.0'
LAST	DC	E'15.0'
STEP	DC	E'0.1'
TABLE	DS	51F
INCR	DC	H'4'
	END	POLY

Name	*Operation*	*Operand*
	LD	6,STEP
LOOP	LD	0,D
	MDR	0,2
	AD	0,C
	MDR	0,2
	AD	0,B
	MDR	0,2
	AD	0,A
STORE	STD	0,TABLE
	CDR	2,4
	BC	8,OVER
	ADR	2,6
	LH	10,STORE+2
	AH	10,INCR
	STH	10,STORE+2
	BC	15,LOOP
OVER	EOJ	
	DS	0D
A	DS	D
B	DS	D
C	DS	D
D	DS	D
FIRST	DC	D'10.0'
LAST	DC	D'15.0'
STEP	DC	D'0.1'
TABLE	DS	51D
INCR	DC	H'8'
	END	POLY

Discussion

When two instructions perform the same function, their only difference being that one has the RR format and the other has the RX format, the one with the RR format not only requires less memory space but also is executed more rapidly. Thus in repetitive calculations such as illustrated in this example it is desirable to keep the quantities most heavily used in the registers, with priority being assigned in order of usage. The program then starts by loading three of the registers, leaving the fourth available for the quantity being computed.

The polynomial evaluation itself not only saves time and storage by use of the factored form but is carried out in a simply structured manner. The form lends itself, therefore, to generalization and is convenient to use for evaluation of polynomials of arbitrary degree.

Regardless of which mode of arithmetic is used for computation, memory addresses are binary integers, and modification of memory addresses makes use of binary arithmetic. The halfword instructions are especially convenient for this type of operation.

In these Solution Programs the storing of the computed value of the polynomial is followed by the end-of-loop test and then by the address modification step. Some programmers prefer to write their loops with the address modification preceding the test, making the program one instruction shorter and consequently more "elegant." The instructions from STORE to the end of the loop are rewritten below to illustrate this technique.

Name	*Operation*	*Operand*
STORE	STE	0,TABLE
	AER	2,6
	LH	10,STORE+2
	AH	10,INCR
	STH	10,STORE+2
	CER	2,4
	BC	13,LOOP

Name	*Operation*	*Operand*
STORE	STD	0,TABLE
	ADR	2,6
	LH	10,STORE+2
	AH	10,INCR
	STH	10,STORE+2
	CDR	2,4
	BC	13,LOOP

Because this approach performs the modification before the test, the contents of Floating-Point Register 2 after the last iteration will be greater than those of Floating-Point Register 4, requiring a different mask in the BC instruction.

EXAMPLE 38

X is a floating-point field. Evaluate the trigonometric sine of X for both short and long operands and store the result in SINEX. Compute the result to five-place accuracy using the infinite series expansion

$$\sin X = X - X^3/3! + X^5/5! - X^7/7! + \cdots$$

Solution Procedure

1. By adding or subtracting multiples of 2π, bring the value of X into the range $-2\pi < X < 2\pi$. This produces the same value for the sine because of the periodic nature of this function but does so with computation of fewer terms.
2. Load Floating-Point Registers 0, 2, and 6 with 1, 1, and X, respectively. Compute X^2 and store it in SINEX.
3. For each term, compute the denominator and numerator; change the sign of the numerator and divide it by the denominator; and add the quotient into Floating-Point Register 6.
4. Continue until the absolute value of the last term computed is less than .00001; then store Floating-Point Register 6 in SINEX.

Solution Flowchart

In Figure 8.2 the logic of this example is diagrammed to emphasize the iterative loops. The words used to describe the computation and testing are based upon the function performed rather than the specific instructions executed, while the labels on the boxes tie them to the specific programs that follow.

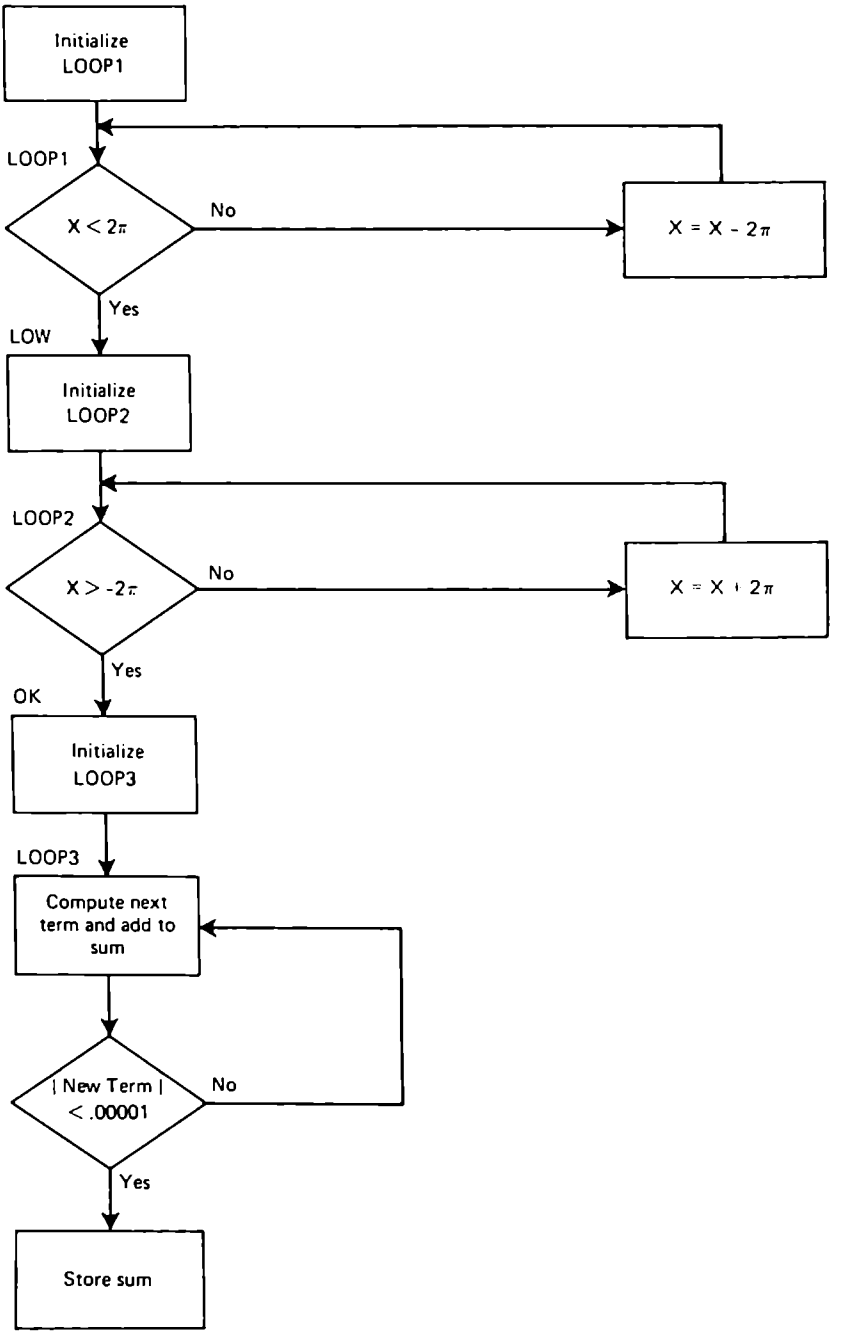


Figure 8.2 Flow chart for Example 38

```
Name    Operation  Operand
        START      8000
SINE    BALR       15,0
        USING      *,15
        LE         2,X
        LE         4,TWOPI
LOOP1   CER        2,4
        BC         4,LOW
        SER        2,4
        BC         15,LOOP1
LOW     LCER       4,4
LOOP2   CER        2,4
        BC         2,OK
        SER        2,4
        BC         15,LOOP2
OK      STE        2,X
        LER        6,2
        MER        2,2
        STE        2,SINEX
        LE         2,ONE
        LER        0,2
LOOP3   AE         2,ONE
        MER        0,2
        AE         2,ONE
        MER        0,2
        LE         4,X
        ME         4,SINEX
        LCER       4,4
        STE        4,X
        DER        4,0
        AER        6,4
        LPER       4,4
        SE         4,TEST
        BC         11,LOOP3
        STE        6,SINEX
        EOJ
        DS         0F
X       DS         F
SINEX   DS         F
ONE     DC         E'1'
TWOPI   DC         E'6.283185'
TEST    DC         E'.00001'
        END        SINE
```

```
Name    Operation  Operand
        START      8000
SINE    BALR       15,0
        USING      *,15
        LD         2,X
        LD         4,TWOPI
LOOP1   CDR        2,4
        BC         4,LOW
        SDR        2,4
        BC         15,LOOP1
LOW     LCDR       4,4
LOOP2   CDR        2,4
        BC         2,OK
        SDR        2,4
        BC         15,LOOP2
OK      STD        2,X
        LDR        6,2
        MDR        2,2
        STD        2,SINEX
        LD         2,ONE
        LDR        0,2
LOOP3   AD         2,ONE
        MDR        0,2
        AD         2,ONE
        MDR        0,2
        LD         4,X
        MD         4,SINEX
        LCDR       4,4
        STD        4,X
        DDR        4,0
        ADR        6,4
        LPDR       4,4
        SD         4,TEST
        BC         11,LOOP3
        STD        6,SINEX
        EOJ
        DS         0D
X       DS         D
SINEX   DS         D
ONE     DC         D'1'
TWOPI   DC         D'6.283185'
TEST    DC         D'.00001'
        END        SINE
```

Discussion

The instructions down to OK adjust the value of X so that, whatever its initial value, it is in the range, $-2\pi < X < 2\pi$. If $X > 2\pi$, 2π is subtracted repeatedly

until this is no longer the case: and if $X < -2\pi$, 2π is added repeatedly until the desired range is attained.

The instructions from OK to LOOP3 set the initial conditions for the term by term evaluation of the infinite series. The adjusted value of X is stored in X; its square is calculated and stored in SINEX: and 1 is loaded into Floating-Point Registers 0 and 2.

LOOP3 starts with four instructions which both compute the denominator of the next term and also retain that information needed to compute the subsequent factorials in the next iteration. The next four instructions compute the new numerator by multiplying the old numerator by the square of X, then change its sign and store it away for use in the next iteration. By reversing the sign this way in each iteration, the alternating signs of the series expansion are easily handled. The next two instructions divide the new numerator by the new denominator to form the new term, and then add its value to Floating-Point Register 6, where the sum of the series is being accumulated. Finally, the new term is tested to determine whether another iteration is necessary or whether the result should be stored in SINEX.

This example illustrates a method of calculation applicable to series expansions of many kinds. More than that, it indicates the feasibility when using a digital computer of computing elementary and other functions as they are required, in contrast to looking them up in a table. Note too that the above program could serve for the computation of cosine X using the series expansion:

$$\cos X = 1 - X^2/2! + X^4/4! - X^6/6! + \ldots$$

with only some minor changes in the initialization steps between OK and LOOP3.

PROBLEMS

1. In what ways does the use of floating-point arithmetic instructions aid or hinder the programmer?
2. What advantage, if any, is there in using the hexadecimal number system in floating-point arithmetic?
3. What do the techniques used for computing square root and evaluating the trigonometric sine have in common?
4. How can the factored form used for polynomial evaluation be incorporated into a subroutine that evaluates polynomials of arbitrary degree?
5. Iterative loops have four basic components: initialization (setting of initial values), computation, modification, and testing. In what orders may these components be arranged?

A, B, C, and D are floating-point fields in memory. In each of the following exercises, use these fields as needed and write both short operand and long operand programs, if appropriate, to produce the specified result.

6. $Y = (A^2 + B^2)/(C^2 + D^2)$
7. $Y = (A^2 + B^2)/(C^2 - D^2)$
   If $C^2 = D^2$, set $Y = 9999 \times (10)^{20}$

8. $Y = 1/(A + 1/(B + 1/(C + 1/(D))))$
9. $Y = A + A^2 + A^3 + \ldots + A^{20}$
10. Compute the 20th term of the sequence

$$1, 1/2, 1/4, 1/8, \ldots$$

Set Y to -1, 0, or 1 according to whether the 20th term is less than, equal to, or greater than 10^{-6}, respectively.

11. Compute the sum of 30 terms of the series

$$1 - 1/2 + 1/3 - 1/4 + \ldots$$

and store the result in Y.

12. Compute the cosine of A using the following formula:

$$\cos A = 1 - A^2/2! + A^4/4! - A^6/6! + \ldots$$

Use as many terms as necessary to make the result correct to six decimal places: that is, keep adding terms until the magnitude of the last is less than 10^{-6}.

13. Compute the natural logarithm of B to nine decimal places using the following formula:

$$\log (1 + B) = B - B^2/2 + B^3/3 - B^4/4 + \ldots$$

14. Compute the inverse hyperbolic sine of C to ten decimal places using the following formula:

$$\sinh^{-1} X = X - \frac{1}{2}\cdot\frac{X^3}{3} + \frac{1\cdot 3}{2\cdot 4}\cdot\frac{X^5}{5} - \frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\cdot\frac{X^7}{7} + \cdots$$

15. Compute one root of the general cubic equation

$$AX^3 + BX^2 + CX + D = 0$$

to five decimal places by repetitive evaluation of the polynomial

$$Y = AX^3 + BX^2 + CX + D$$

and by application of the recursive relationship

$$X_{i+2} = (X_i Y_{i+1} - X_{i+1} Y_i)/(Y_{i-1} - Y_i)$$

where Y_i is the value of Y when $X = X_i$, etc. [*Hint:* To get the loop started, first choose X_1 and X_2 arbitrarily and compute Y_1 and Y_2. This provides initial values for use of the recursion formula.

Chapter 9

Logical Instructions

In addition to the three sets of arithmetic instructions which treat data as numerical quantities, System/360 includes a wide variety of other instructions useful in manipulating data but which do not necessarily treat such data as numbers. Some of these have already been encountered; for example, Move, Move Numerics, and Move Zones. Others are included in this chapter. All of these are often called *logical* instructions.

There are close relationships between certain of the logical instructions and the operations of set theory. The instructions are called AND, OR, and EXCLUSIVE OR, and the relationships can be shown by means of Venn diagrams.

Figure 9.1 illustrates the AND instructions by schematically depicting two sets, A and B, and their intersection, the shaded area. The AND instructions match two sets of bits. Wherever corresponding bits are both 1, this is considered an intersection and the resultant bit is made 1. Wherever either or both of the corresponding bits is or are 0, the resultant bit is made 0.

Figure 9.2 illustrates the OR instructions by depicting the union of two sets A and B. The OR instructions match two sets of bits. Wherever either or both of corresponding bits is or are 1, this is considered a union and the resultant bit is made 1. Whenever both of the bits are 0, the resultant bit is made 0.

Figure 9.3 illustrates the EXCLUSIVE OR instructions by depicting as the shaded area the part of the union that is not also part of the intersection.

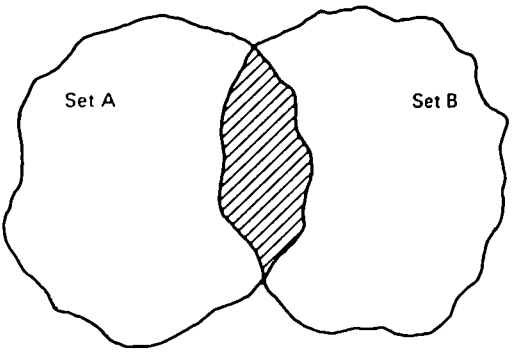


Figure 9.1 Illustration of the AND instructions

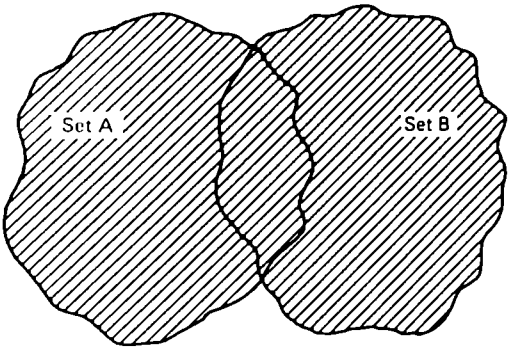


Figure 9.2 Illustration of the OR instructions

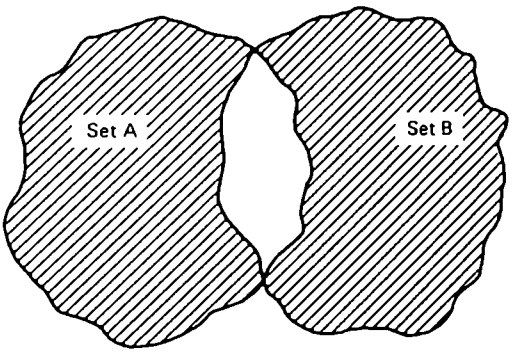


Figure 9.3 Illustration of the EXCLUSIVE OR instructions

The EXCLUSIVE OR instructions match two sets of bits. Whenever a pair of corresponding bits consists of a 0 and a 1, the resultant bit is made 1. When both are 1 or both are 0, the resultant bit is made 0.

Others of the logical instructions resemble many of the instructions previously discussed. In total, their impact is to give the programmer the power to treat a wide variety of both straightforward and subtle programming requirements in a direct manner.

9.1 THE AND INSTRUCTIONS

The System/360 Assembler Language repertoire includes four instructions for performing the AND operation. Although each of these allows the same basic type of bit-by-bit masking, the choice gives the programmer a full variety of operand types. Regardless of which AND instruction is used, this operation presents a very convenient way to set a desired bit position to 0 while leaving others unchanged.

9.1.1 The NI Instruction

AND Immediate (NI) allows the "ANDing" of a byte in memory and a one-byte mask specified in the instruction itself. If corresponding bits in the two operands are 1's, that bit position in the first operand will remain a 1. For all other contingencies it will be set to 0. When all eight pairs of bits have been compared this way the condition code will be set to 00 (table value of 8) if the result is all 0's; otherwise it will be set to 01 (table value of 4). For example, if we defined

```
LOOK   DC     X'C5'
```

and then wrote

```
       NI     LOOK,X'07'
       BC     8,NOONE
       BC     4,ONES
```

the comparison would be as follows:

LOOK	1	1	0	0	0	1	0	1
mask	0	0	0	0	0	1	1	1
resulting LOOK	0	0	0	0	0	1	0	1

Hence the presence of the 1's in LOOK after execution will set the table value to 4 and the branch to ONES will be taken.

9.1.2 The NR Instruction

This AND instruction allows the same kind of logical operation to be performed on the contents of two general registers. As before, the result is stored in the first operand and the condition code is set as described for the NI.

Thus, for example, the definitions

```
TRY1    DC    X'D504009E'
TRY2    DC    X'F1306700'
```

and the sequence

```
L       9,TRY1
L       11,TRY2
NR      9,11
BC      8,NOT
BC      4,GOT
```

will produce

```
Register 9          11010101000001000000000010011110
Register 11         11110001001100000110011100000000
new Register 9      11010001000000000000000000000000
```

so that the branch to GOT will be taken.

9.1.3 The N Instruction

The RX version of the AND operation works the same way as NR, except that the second operand refers to a fullword in storage. With TRY1 and TRY2 defined as above, the sequence

```
L       9,TRY1
N       9,TRY2
BC      8,NOT
BC      4,GOT
```

will produce the same results as before.

9.1.4 The NC Instruction

The AND operation may be performed on as many as 256 bytes with one instruction by using the SS version NC. Both operands must be the same length. Referring again to TRY1 and TRY2, the sequence

```
NC    TRY1,TRY2
BC    8,NOT
BC    4,GOT
```

will once again cause the branch to GOT to be executed, but this time TRY1, and not a register, will now contain the result of the AND operation.

EXAMPLE 39

An SS instruction whose name is SET is part of a certain procedure. As part of the address modification required in that procedure, the displacement of the first operand must be set to zero. Write a program to do this, making sure that the base register address is not altered.

Solution Procedure

This program can be written using previously introduced instructions. The middle halfword of SET could be loaded into a general register, shifted right twelve positions to eliminate the displacement, shifted left twelve positions to produce the desired result, and restored to its place in SET. Using logical instructions, the same effect is produced more simply.

1. Load the middle halfword of SET into a general register.
2. Use an AND instruction with an appropriate mask to set the displacement to zero without alteration of the base register address.
3. Store the result back into SET.

Solution Program

```
Name    Operation   Operand
        START       10000
AND     BALR        15,0
        USING       *,15
        LH          10,SET+2
        N           10,MASK
        STH         10,SET+2
          .                .
          .
        EOJ
        DS          0F
MASK    DC          XL4'FFFFF000'
        END         AND
```

Discussion

This Solution Program, being an intialization procedure, is only a part of a program, and the gap between STH and EOJ indicates that the presence of other instructions is assumed. Among these other instructions is the SS instruction named SET.

The LH instruction selects the middle halfword of SET, expands it into a fullword,

and loads it into General Register 10. The low-order half of this register, then, contains the base register address and the displacement of SET's first operand after execution of the LH. An alternate statement that appears to produce the same result is

```
L     10,SET
```

This latter approach, however, has a defect. The second operand address of L must be divisible by 4, and there is no way to insure that this will be so. The halfword approach sidesteps this sort of difficulty because all instructions have addresses divisible by 2. By setting up a mask of FFFFF000 (twenty 1's and 12 0's) we can treat it as an instruction with a zero displacement. Thus the last 12 bits of Register 10 will be set to 0's and the others will remain unchanged by the AND operation.

This example could have been programmed in other ways as well. One alternate is to replace LH, N, and STH by the following two instructions:

```
NI    SET+2,240
NI    SET+3,0
```

Since NI operates on only one byte, it takes two instructions to set the 12 bits of SET's first operand to zero. The first NI specifies a mask of 240_{10} (11110000_2). This takes care of the four high-order bits in the displacement. The second NI zeros the remaining eight bits.

9.2
THE OR INSTRUCTIONS

The four available instructions for "ORing" two operands together are directly analogous to the AND instructions described in Sec. 9.1. When corresponding bits in the first and second operands are both 0's, the bit in the first operand is left as zero; for all other contingencies that bit is set to 1. Thus, while the AND instructions provide a convenient way of using a mask to set certain bits to zero without affecting others, the OR instructions can be used in conjunction with an appropriate mask to set certain bits to 1's.

The SI, RR, RX, and SS versions have respective operation codes of OI, OR, O, and OC. Condition codes are set as for the AND instructions.

EXAMPLE 40

Just as it may be required to set bits to 0 on a selective basis, so it may also be necessary to set bits to 1 selectively. In this example, write an instruction that sets the length field of an NC instruction named SELECT to 256.

Solution Procedure

Recall that an eight-bit length field represents lengths of from 1 to 256 bytes by means of the binary integers 0 to 255, respectively. Therefore, a length of 256 bytes is represented by the binary equivalent of 255, a field of eight 1's. Since the length field of an SS instruction occupies bits 8–15, it is the second byte, or SELECT+1 in this case.

Solution Program

```
Name    Operation   Operand
NAME    OI          SELECT+1,255
```

9.3 THE EXCLUSIVE OR INSTRUCTIONS

XI, XR, X, and XC form the third set of System/360's basic logical instructions. Their form and operation is essentially the same as for the AND and OR sets, but the logical ground rules are as follows: When corresponding bits in the two operands match (both are 1's or both are 0's) the bit in the first operand is set to 0; otherwise it is set to 1. Thus if we define

```
LOOK1   DC      X'F6'
```

and we write

```
        XI      LOOK1,X'3E'
```

the operation will be

LOOK1	1	1	1	1	0	1	1	0
Mask	0	0	1	1	1	1	1	0
new LOOK1	1	1	0	0	1	0	0	0

XR, X, and XC operate the same way. The condition code is set as before.

EXAMPLE 41

A, B, and C are three floating-point doublewords in memory. Set the sign of A to plus, set the sign of B to minus, and reverse the sign of C.

Solution Procedure

The requirements of this example may be carried out with the Load Positive, Load Negative, and Load Complement instructions of the previous chapter. Instead, however, we shall use the following logical instructions:

1. AND to set the sign of A to plus (0).
2. OR to set the sign of B to minus (1).
3. EXCLUSIVE OR to reverse the sign of C.

Solution Program

```
Name    Operation   Operand
        START       8000
SIGN    BALR        15,0
        USING       *,15
        NI          A,127
        OI          B,128
```

```
Name    Operation   Operand
        XI          C,128
        EOJ
        DS          0D
A       DS          D
B       DS          D
C       DS          D
        END         SIGN
```

Discussion

The 127 (base 10) in the first instruction is equal to 01111111 (base 2). Applied to A via the NI, therefore, it sets the sign bit to 0 (plus) and has no effect on any other bits. Similarly, the 128 (base 10) of the second instruction is equal to 10000000 (base 2). As the second operand of OI, it sets the sign bit of B to 1 (minus) and has no effect on any other bits.

As in the OI, the second operand of XI in this example is equal to 1000000 (base 2). If the sign bit of C is originally 1, it will be changed to 0 because the corresponding bits are the same; if the sign bit of C is originally 0, it will be changed to 1 because the corresponding bits are not the same; thus, applying a 1 with an EXCLUSIVE OR always changes a 1 to a 0 and a 0 to a 1. The 0 bits of the second operand have no effect on the first operand because a 0 applied to a 0 leaves it 0, and a 0 applied to a 1 leaves it 1, according to the above rule.

EXAMPLE 42

Write a program to exchange the contents of two general registers without using a third register or memory as an intermediate repository.

Solution Procedure

1. Use EXCLUSIVE OR to place the modulo-two sum of the operands in the first register.
2. Use EXCLUSIVE OR to place the modulo-two sum of the operands in the second register.
3. Use EXCLUSIVE OR to place the modulo-two sum of the operands in the first register.

Solution Program

```
Name      Operation   Operand
          START       8000
CHANGE    BALR        15,0
          USING       *,15
          XR          10,11
          XR          11,10
          XR          10,11
          EOJ
          END         CHANGE
```

TABLE 9.1 STEPWISE ANALYSIS OF EXAMPLE 42

10	01101011110010001010110100011101111	At start
11	00000000110101001001100100111001010	
10	01101011101000101111000111011011101	After first XR
11	00000000110101001001100100111001010	
10	01101011010001011111000111011011101	After second XR
11	01101011110010001010110100011101111	
10	00000000110101001001100100111001010	At conclusion
11	01101011110010001010110100011101111	

Discussion

There are several ways to exchange the contents of two general registers. This one not only is unusual in requiring neither auxiliary registers nor memory but also provides additional insight into the EXCLUSIVE OR operation. In this Solution Program, the contents of General Registers 10 and 11 are exchanged.

EXCLUSIVE OR, like AND and OR, operates on corresponding bits of the operands and replaces the first operand bit with the result. There is no effect on this operation caused by adjacent bits, as with carries in arithmetic. This Solution Program may thus be analyzed by examining what happens in a single bit-position as the XR instructions are applied. The results of such an analysis are summarized in Table 9.1 for sample values.

9.4 THE STC AND IC INSTRUCTIONS

This pair of RX instructions provides the Assembler Language programmer with the ability to move a single byte between storage and one of the general registers. Transfer from storage to register is handled by

name IC *n,oprnd*

which results in the single byte at the location named *oprnd* being duplicated in the rightmost eight bits of General Register *n*. The rest of the register is undisturbed.

Transfer in the other direction (from register to memory) is handled by

name STC *n,oprnd*

which duplicates bits 24–31 of register *n* in the location named by *oprnd*. Neither STC or IC affects the condition code.

EXAMPLE 43

FIELD is a field of 256 characters in memory, and NUMBER is a positive binary integer whose value is not greater than 256. Write a program to clear to zero as many bytes of FIELD as specified by NUMBER.

Solution Procedure

1. Reduce the magnitude of NUMBER by one so that its low-order byte may serve as the length field of an XC instruction.
2. Insert the low-order byte so generated in an XC instruction.
3. Execute the XC instruction to produce the required zero bytes.

Solution Program

```
Name     Operation  Operand
         START      8000
ZEROS    BALR       15,0
         USING      *,15
         L          10,NUMBER
         S          10,ONE
         STC        10,*+5
         XC         FIELD,FIELD
         EOJ
FIELD    DS         CL256
         DS         0F
NUMBER   DS         F
ONE      DC         F'1'
         END        ZEROS
```

Discussion

The magnitude of NUMBER is reduced by one because an eight-bit length field represents lengths of 1-256 by means of the binary equivalents of 0-255, respectively. The resultant integer is stored in the length field of the XC instruction using STC.

In this Solution Program the second operand of STC is *+5. Recall that an asterisk used in this manner is interpreted as the address of the instruction in which it appears. The relative address *+5, therefore, designates a byte five bytes beyond the first byte of STC. Because STC has the RX format and XC has the SS format, *+5 references the length field of the latter instruction. This is shown schematically as follows:

STC	*	*+1	*+2	*+3		
XC	*+4	*+5				

Another way to produce the same result, of course, is to give the XC statement a name, say NAME, and address its length field as NAME+1. This approach is less

desirable in large programs in which the limit on the number of symbols might be exceeded, but otherwise might be preferred for clarity.

At assembly time, the length field of the XC instruction in the Solution Program is set to 1's because the defined length of the first operand is 256 bytes. Before this instruction is executed, however, it is appropriately modified by the preceding instructions. Writing this instruction with identical operands produces the desired 0's in the result because corresponding bits are always both 0's or 1's. EXCLUSIVE OR sets the result of such combinations to 0's (and of 0–1 and 1–0 combinations to 1's).

9.5 LOGICAL ADDITION AND SUBTRACTION

The four instructions for logical addition and subtraction operate on fullwords, treating them as 32-bit integers rather than 31 bits plus a sign. Overflow, as can occur in ordinary addition or subtraction (that is, with A or S) is, by definition, avoided. Instead, the resulting sum or difference is placed in the register specified in the first operand and the condition code is set as follows:

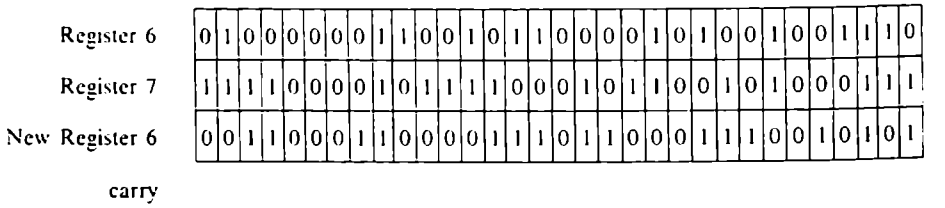
Condition Code	*Table Value*	*Set When*
00	8	Result = 0, no carry
01	4	Result ≠ 0, no carry
10	2	Result = 0, carry
11	1	Result ≠ 0, carry

The RX versions for logical addition and subtraction have mnemonic codes of AL and SL, respectively; corresponding RR versions are ALR and SLR.

As an example, let us examine the results of the statement

```
ALR    6,7
```

with the contents of Registers 6 and 7 as shown.



If we go through the addition, it is seen that the result is nonzero, but there is a carry. Consequently, the condition code will be set to 11. Thus, the programmer can use logical addition before he attempts regular addition in anticipation of an overflow, so that appropriate steps can be taken to avoid it.

9.6
THE LOAD ADDRESS INSTRUCTION

This RX instruction will place the *address* of the second operand into the register specified in the first operand. Thus

```
LA      5,LOOK
```

will place LOOK's address in Register 5. It can also be used to store literal values. If we were to write

```
LA      5,1201
```

for example, the binary equivalent of $(1201)_{10}$ would be generated in Register 5.

By specifying an index register of zero, it is possible to use the LA instruction for incrementation. The following sequence illustrates this:

Statement		*Resulting Contents of Register 3*
LA	3, 50	50
LA	3, 5(0,3)	55
LA	3, 5(0,3)	60
LA	3, 12(0,3)	72
LA	3, 81	81
LA	3, 2(0,3)	83

EXAMPLE 44

Write a subroutine which makes available a pseudorandom number between 1 and 1000 each time it is called. We shall use the given linkage

```
BAL     11,RANDOM
```

Solution Procedure

Define TABLE as a 50-word array, and assume these words have previously been filled with random numbers taken from a handbook and entered via an input device.

1. Compute each pseudorandom number by adding the previously formed number, which has been placed in TABLE, to the number following it in the array. Replace the latter number by the new one. The first element of the array is considered to follow the last, making the array endless.
2. Set to zero the 22 high-order bits of the new pseudorandom number generated. Those that remain are in the range 0-1023. Add 1 to the result to make the range 1-1024.
3. Prepare for the calculation of the next pseudorandom number by adding four to the addresses which refer to the array and by subtracting two hundred from these addresses when the end of the array has been reached.
4. If the number calculated is in the range 1001-1024, return to the beginning and repeat; if it is in the range 1-1000, exit from the subroutine.

Solution Program

```
Name      Operation  Operand          Comment

          .          .
          BAL        11,RANDOM

          .
RANDOM    L          10,TABLE
          AL         10,TABLE+4
          ST         10,TABLE+4       STORE NEW NUMBER
          N          10,MASK          ADJUST TO 0-1023
          AH         10,ONE           ADJUST TO 1-1024
          LH         8,RANDOM+6
          STH        8,RANDOM+2       MODIFY ADDRESS OF L IN-
                                      STRUCTION
          AH         8,FOUR
          STH        8,*+6
          LA         9,0(0,0)
          C          9,ADCON          CHECK FOR END OF ARRAY
          BC         7,AROUND
          SH         8,H200
AROUND    STH        8,RANDOM+6
          STH        8,RANDOM+10
          C          10,THOUS         CHECK FOR NEW NUMBER >
                                      1000
          BC         2,RANDOM         REPEAT IF IT IS
          BCR        15,11
          DS         0F
TABLE     DS         50F
MASK      DC         XL4'3FF'
ONE       DC         H'1'
FOUR      DC         H'4'
ADCON     DC         A(TABLE+200)
THOUS     DC         F'1000'
H200      DC         H'200'
          .
```

Discussion

The method used here for generation of pseudorandom numers is only one of many and was chosen primarily for its value in illustrating instructions and programming techniques.

Each time the subroutine is entered at RANDOM, the first five instructions compute a new pseudorandom number, store it in the table for use the next time, and place it in the range 1-1024. Subsequently, the second C/BC pair initiates a

recalculation whenever the number is greater than 1000. Computation of the new pseudorandom number is performed by a single addition, an operation that normally requires no special consideration. In this case, however, the addition is performed each time the subroutine is entered, and this may occur tens of thousands of times during the course of a run. If an arithmetic ADD were used, sooner or later an overflow condition would arise and interrupt processing. Consequently, the AL instruction is used instead.

Most of the other instructions in the example are for the purpose of address modification. The second operands of the first three instructions must be incremented by four on each iteration and must be decremented by two hundred whenever they extend beyond the range of TABLE. The LH/STH pair modifies the second operand of RANDOM in a direct way, using the fact that whatever address had been used in the following two instructions is now appropriate for it. Modifying these following two instructions is not as direct, however. Ordinarily, an increment of four is sufficient, but each time a test must be made to determine when the address is to be reset to the beginning of the array. This test is carried out by the sequence

```
STH     8,*+6
LA      9,0(0,0)
C       9,ADCON
```

STH stores the low-order half of General Register 8 (which contains a four-bit base register address and twelve-bit displacement) in *+6. This is within LA, the next instruction. At assembly time, the LA instruction used in this Solution Program contains zeros for the index and base register addresses and for the displacement. When this instruction is executed, however, it still contains zero as the index register address but also contains the nonzero base register address and displacement stored there by the STH instruction. The function of this instruction in this example, then, is to compute and place in General Register 9 the modified second operand addresses of the second and third instructions of the subroutine. Once there, it is compared with ADCON, defined as the address two hundred bytes greater than TABLE. The result of this comparison determines whether the contents of General Register 8 are to be reduced by two hundred.

An alternate method of handling the address modification in this example is to use index registers, and an Alternate Solution Program based on this approach is presented below. Note that in either case the subroutine is written in such a way that information is not left in general registers in between executions. Thus the remainder of the program can be written without any special restrictions being placed on the use of general registers. (There are ordinarily restrictions on the use of general registers that arise from some being reserved for the use of the operating system.)

Alternate Solution Program

```
Name      Operation   Operand
                        .
                        .
                        .
RANDOM    LM          4,7 INDEX1
```

```
Name      Operation   Operand
          L           10.TABLE(4)
          AL          10.TABLE(5)
          ST          10.TABLE(5)
          N           10.MASK
          AH          10.ONE
          ST          5.INDEX1
          BXLE        5.6.BRANCH
          SR          5.5
BRANCH    ST          5.INDEX2
          C           10.THOUS
          BC          2.RANDOM
          BCR         15.11
          DS          0F
INDEX1    DC          F'0'
INDEX2    DC          F'4'
FOUR      DC          F'4'
          DC          F'196'
TABLE     DS          50F
MASK      DC          XL4'3FF'
THOUS     DC          F'1000'
ONE       DC          H'1'
```

. . .

9.7 THE LOGICAL SHIFT INSTRUCTIONS

There are four logical shift instructions whose operations parallel those described for the algebraic shifts discussed previously (Sec. 5.7 and 5.8). All four are of the RS type and the codes are as follows:

Code	*Name*
SLL	Shift Left Single
SLDL	Shift Left Double
SRL	Shift Right Single
SRDL	Shift Right Double

The logical shift instructions act like their algebraic counterparts except that they do not treat the contents of a general register as a number plus sign. Instead, the entire register is shifted by the number of bits indicated in the second operand. For example, if we wrote

```
SLL     4,3
```

the contents (all 32 bits) of Register 4 would be shifted three positions to the left. The leftmost three bits would be lost and zeros would be used to

fill bits 29–31, regardless of the value of the high-order bit. (Recall that the filler for the SLA is determined by the value of the sign bit.)

The double shift treats the register specified in the first operand as the first (high-order) half of a 64-bit field (the register with the next higher address being the implied second half) and all 64 bits are shifted as a unit. An even-odd pair must always be specified. Thus

```
SRDL    8,5
```

will shift the 64 bits in Registers 8 and 9 over to the right by five positions, filling positions 0–4 in Register 8 with zeros. None of the logical shift instructions affects the condition code.

EXAMPLE 45

PACKED is the name of a word that consists of five segments: A, B, C, D, and E. Write a program to extract the individual segments and place them in specific positions of separate words. Both the original positions and the required positions of the segments are as follows:

Segment	*Original Positions*	*Required Positions*
A	0-1	30-31
B	2-6	22-26
C	7	7
D	8-16	23-31
E	17-31	7-21

Short fields are often packed together as in PACKED to conserve space when tens of thousands of records are stored on magnetic tapes, magnetic disks, or other auxiliary storage media. For use in processing, it is usually necessary to place them in separate bytes, halfwords, or words.

Solution Procedure

1. An even-numbered general register is cleared to zero, and PACKED is loaded into the general register whose address is one greater.
2. A series of shifts of both registers to the left is used to isolate the segments one at a time in the even-numbered register.
3. Segments so isolated are moved into the specified bit positions, primarily by single register shifts, and are stored as words in the required locations.
4. The even-numbered register is cleared to zero after storage of each field to prepare for isolation of the next field.

Solution Program

```
Name    Operation   Operand
        START       9000
SPLIT   BALR        15,0
        USING       *,15
        SR          10,10
```

```
Name      Operation  Operand     Comment
          L          11,PACKED   PACKED IN REG. 11
          SLDL       10,2        0-1 OF PACKED IN 30-31 OF 10
          ST         10,A
          SR         10,10
          SLDL       10,5        SEGMENT B IN 27-31 OF 10
          SLL        10,5        SEGMENT B POSITIONED
          ST         10,B
          SR         10,10
          SLDL       10,1        SEGMENT C IN 31 OF 10
          SLL        10,24       SEGMENT C POSITIONED
          ST         10,C
          SR         10,10
          SLDL       10,9        SEGMENT D IN POSITION
          ST         10,D
          SRL        11,7        POSITION SEGMENT E
          ST         11,E
          EOJ
          DS         0F
PACKED    DS         F
A         DS         F
B         DS         F
C         DS         F
D         DS         F
E         DS         F
          END        SPLIT
```

Note that double-register shifting is not the only way to isolate segments of a word. Another technique is to use the AND instruction to set unwanted segments to zero and to use single-register shifting for positioning. With this technique, C may be isolated and stored by executing the three instructions

```
L     12,PACKED
N     12,MASK
ST    12,C
```

where the mask is defined as XL4'1000000'. Shifting is a relatively time-consuming operation, and a combination of the two isolation techniques is ordinarily used to keep the number of positions shifted to a minimum.

9.8 THE LOGICAL COMPARE INSTRUCTION

It is not sufficient to be able to compare numerical quantities for relative magnitudes. Many occasions call for the comparison of characters, followed by branches based on the outcome. System/360 provides a variety of such comparisons.

9.8.1 The CLC Instruction

The Compare Logical Character (CLC) is an SS instruction causing the character-by-character comparison of two fields in storage. Processing starts from the left and continues for a number of characters implied by the length attribute of the first operand. If the second operand is too short, additional blanks are supplied automatically. The condition code is set as follows:

Condition Code	*Table Value*	*Set When*
00	8	Operands are equal
01	4	First operand is smaller
10	2	First operand is greater

"Smaller" and "greater" refer to the relative position of a character in the collating sequence (see Chapter 2). Thus an A will be "less than" a B, and so forth. CLC is very useful for alphabetic sorting and similar processes.

9.8.2 The CLI, CL, and CLR Instructions

Additional character comparisons can be performed by the CLI, CL, and CLR instructions. CLI is an SI instruction which allows the comparison of a byte in storage with the literal character in the second operand. Thus

```
CLI     B,X'C4'
```

will compare the contents of location B with 11000100.

The RX equivalent is

```
CL      n,oprnd
```

where *n* is a general-purpose register and *oprnd* refers to a place in storage. Similarly, CLR is used for comparing two registers. In all cases, the condition code is set as shown in the previous section for CLC.

9.9 THE EX INSTRUCTION

System/360 provides a most powerful means for instruction modification via the EX instruction. This RX instruction, having the form

```
EX      n,oprnd
```

causes the instruction in *oprnd* to be executed. If that instruction is not a branch, control will be returned to the instruction following the EX. Modification of the instruction in oprnd can be effected as follows:

1. The OR operation is applied to bit positions 8–15 of the instruction at *oprnd* and bits 24–31 of register *n*. The logical sum replaces bit positions 8–15 but only during the execution of the instruction; the second operand is not permanently altered.
2. The instruction at the second operand address is executed. If this instruction alters the condition code, then that becomes the status after completion of EX; EX does not affect the condition code in its own right.
3. After execution of the instruction at the second operand address, control returns to the instruction following EX, and normal sequencing continues.

EX provides a convenient means of indirect length, index, mask, immediate data, and register specification. The instruction at the second operand address may not be another Execute. When General Register 0 is the first operand address, no modification of the second operand takes place.

EXAMPLE 46

INPUT, OUTPUT, and LIST are fields each 256 characters in length. Write a program to move to OUTPUT all those characters of INPUT up to and including the first period. If there is no period in INPUT, insert a period in the first position of OUTPUT. LIST is the table used in conjunction with the TRT instruction.

Solution Procedure

1. Use TRT to obtain the address of the first period in INPUT. If there is none, insert one at OUTPUT.
2. Compute the number of characters to be moved from INPUT to OUTPUT.
3. Use Execute to cause the modification and indirect execution of an MVC instruction to carry out the desired transfer.

Solution Program

```
Name      Operation   Operand
          START       10000
MOVER     BALR        15,0
          USING       *,15
          TRT         INPUT,LIST
          MVC         OUTPUT(1),PERIOD
          BC          8,NEXT
          S           1,ADCON
          EX          1,MOVE
          BC          15,NEXT
INPUT     DS          CL256
OUTPUT    DS          CL256
LIST      DC          XL75'0'
PERIOD    DC          X'4B'
          DC          XL180'0'
          DS          0F
```

```
Name     Operation   Operand
ADCON    DC          A(INPUT)
NEXT
.
.
.
MOVE     MVC         OUTPUT(0),INPUT
```

Discussion

The list (LIST) used in conjunction with TRT consists of all zeros except for a nonzero byte in the 76th position. This position was chosen because the EBCDIC code for the period is 4B (base 16) = 75 (base 10) and because TRT is then able to identify periods by a nonzero byte at the address LIST+75 (this is the 76th byte in the list). Insofar as its usage in TRT is concerned, this nonzero byte may have any nonzero value. The actual period code was selected in this example so that it could also be used in the first MVC instruction.

Execution of TRT places the address of the first period in General Register 1 if one is found, or it sets the condition code to 00 if a period is not found. Then, MVC places a period in OUTPUT, as if a period has not been found. (This period is subsequently overlaid if there is, indeed, a period within INPUT.) Because MVC does not affect the condition code, the BC tests it as it was left by TRT. If no period was found, the branch to NEXT is taken, and OUTPUT remains as set; if a period was found, however, control progresses to the next instruction in sequence.

The Subtract instruction reduces the address in General Register 1 by the address of INPUT, leaving as the difference a quantity whose value is one less than the number of characters to be moved from INPUT to OUTPUT. Because length fields require quantities to be one less than the number of characters involved in the operation, this result is exactly what is needed for the subsequent MVC instruction.

Although the computed length field could be inserted in an MVC instruction and the MVC instruction could be executed, both of these operations are performed instead by the EX. This instruction causes the length field of the second operand to be set to the proper value for the number of characters to be moved, produces the movement of the characters through execution of the MVC, and then returns control to the unconditional branch to NEXT.

PROBLEMS

1. What characteristics do the AND, OR, and EXCLUSIVE OR instructions have in common?
2. What are the fundamental differences between the arithmetic and logical instructions for addition, subtraction, and shifting?
3. What is the difference between the results obtained by the following two instructions

```
L     5,100(3,4)
LA    5,100(3,4)
```

4. Assume that it is necessary to place a byte in a specific location in memory. When would STC be used in preference to MVC?
5. Assume that the Translate and Test instruction is to be used to locate all the punctuation in a long character field. How might the Execute instruction be used to facilitate this process?

An 800-byte field in memory contains numeric, alphabetic, and special characters. Use logical instructions as applicable to carry out the following functions:

6. Compute a 32-bit "hash total" by considering each group of four bytes a word and adding them together.
7. Set A = 1 if the hash total computed above is equal to A1A1A1A1 (base 16); otherwise, set A = 0.
8. Set the sixth bit in each of the 800 bytes to zero.
9. Set the eighth bit in each of the 800 bytes to one.
10. Invert the third bit of each byte.
11. "Rotate" the 800-byte field 50 bits to the right; that is, shift the contents 50 positions to the right and insert on the left the 50 bits shifted off the low-order end.
12. Set the first 400 bytes to zero and the second 400 to one.
13. In the original 800-byte field, find the address of the third comma. Set A = 1 if the third comma is in the 125th byte; otherwise, set A = 0.
14. Count the number of bit-positions in the 800-byte field that contain a 1.
15. Compute the correct arithmetic sum and difference of two positive binary integers each 100 bytes long.

Chapter 10

Input/Output

The extensive input/output versatility provided by System/360 through the use of channels (see Chapter 1) carries with it the unavoidable penalty of relatively complicated data transmission processes. It can no longer be a simple matter of reading from or writing on a specified device. Instead, elaborate preparations must be undertaken to ensure the readiness of the device and the channel to which it is attached, as well as that of the central processor. When an I/O operation is concluded, a similar process must ensue to inform various system components of this.

If the fulfillment of these obligations were left to the applications programmer, it would mean that he would be forced to devote a disproportionate amount of his effort to preparations surrounding I/O operations, often dwarfing the problem program itself. Consequently, System/360 is designed to remove this responsibility from the problem programmer's domain. In fact, it has been made extremely difficult and awkward for him to try to assume this burden.

The basic strategy for achieving this was to design the actual I/O operations as privileged instructions, executable only when the processor is in the supervisor state. As we saw in Chapter 3, this essentially places such instructions out of the reach of the programmer, since he has immediate access only to those instructions operable in the problem state. Consequently, the programmer's basic tasks consist of initiating and terminating I/O operations and providing the supervisory software with the necessary information for actual execution.

In order to understand the interaction between the problem program and supervisor routines with regard to I/O operations, we shall examine the basic sequence of events in a data-transmission process. For this purpose it will not be necessary to distinguish the programmer's duties from those of the system.

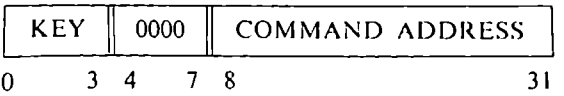
10.1 BASIC I/O PROCESSES

Suppose some sort of computation, such as any of the examples in the previous chapters, has been taking place, and the program has reached a point where an input or an output operation is required. A typical input operation might be reading a card containing additional data; a typical output operation might be printing a line of processed results. The program of instructions in the CPU must "notify" the channel to be used for the input/output operation that it is to begin its program. (Recall that whereas a CPU program is said to consist of instructions, a channel program is said to consist of commands— see Sec. 3.4.2.1. Commands, like instructions, are stored in memory. Unlike instructions, however, there are only six, and they have their own format.)

The CPU program notifies the channel program that it is to start by executing the SIO (Start I/O) instruction. SIO is an SI type instruction with a single operand. Although this operand is written, either symbolically or absolutely, in terms of a base register and displacement, it is an operand only in the sense that the second operand in a shift instruction is an operand. Bit positions 16-23 of the effective address identify the channel number, and bit positions 24-31 identify the input/output device to be used. SIO does not itself contain any of the other information necessary for the operation— information such as the type of operation, the address of memory locations where input data are to be stored or output results obtained, the quantity of data to be moved. Remember that the prime function of SIO is to notify a channel to start its program.

10.1.1 The Channel Address Word

The execution of SIO initiates a chain of events that is dependent upon appropriately chosen information already having been placed in memory. The first event is the movement of the Channel Address Word (CAW) from bytes 72-75 of memory to the channel designated by SIO. The CAW, which contains the memory address of the first command of the channel program, (see Sec. 3.4.2.1) has the following format:



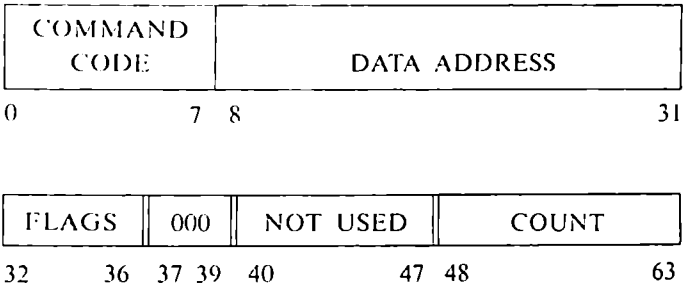
When the storage protection feature is installed in a System/360, and when the key field is nonzero, the latter is used as a check to ensure that information being entered does not inadvertently destroy information already in memory. This check is carried out by matching the key field of the CAW against the key previously assigned to the memory area affected. (For this purpose, each 2,048-byte block of memory is assigned a key.) In addition to this storage-protection feature, a further feature that protects against information being inadvertently fetched from memory uses the same key-matching scheme when storage protection is installed. Note that the key field of the current PSW plays the same role with respect to CPU programs as the key field of the CAW plays with respect to channel programs.

When neither protection feature is installed, or when such a feature is installed but not used, bit positions 0–3 of the CAW must be zero. Bit positions 4–7 of the CAW, though not used, must also be zero.

The command address in bit positions 8–31 of the CAW is the effective address of the first (there might be only one) command in the channel program. Commands are stored in memory in the same manner as instructions but do not have the same formats.

10.1.2 The Channel Command Word

Commands are doublewords, each of which is embodied in a 64-bit fixed-format Channel Command Word (CCW). These are normally executed consecutively, but branches are also possible. The format of a CCW is as follows:



10.1.2.1 Channel Commands

A channel can execute six types of commands, and these are specified in their various modified forms in bit positions 0–7 of each CCW. The six types of commands are Read, Read Backward, Sense, Write, Control, and Transfer In Channel. Read causes the transfer of information from an input device to memory. Read Backward is a special command used only to read information from a magnetic tape moving in the direction opposite to that in which it was recorded. Sense also transfers information to memory.

but device status information rather than program information. Write causes the transfer of information from memory to an output device. Control is used for device setup, such as rewinding a reel of magnetic tape or positioning the access arm of a magnetic disk drive. Transfer In Channel is a branch instruction.

10.1.2.2 Data Address

When data are transferred to or from memory, the memory address is given in bit positions 8–31 of the CCW, and the number of bytes to be transferred is given in bit positions 48–63. For the Transfer In Channel command, the branch address is given in bit positions 8–31.

10.1.2.3 CCW Flags

A number of the options available in channel programs are specified by means of the five flags in bit positions 32–36 of each CCW. These are called, respectively, the Chain-Data Flag (CD), the Chain-Command Flag (CC), the Suppress-Length-Indication Flag (SLI), the Skip Flag (SKIP), and the Program-Controlled-Interruption Flag (PCI).

When CD is 1, *data chaining* is specified. This means that an input/output operation that has transferred all the data specified by a CCW is to continue the operation according to the data specifications of the next consecutive CCW (the next doubleword in memory). Thus a given input/output operation is not constrained to place data in or retrieve results from consecutive locations in memory. When CD is zero, data chaining does not occur.

When CC is 1, *command chaining* is specified. This means that the input/output operation specified in the following CCW is to be treated as a new operation and to be executed next. When neither chaining flag is 1, the current CCW is the last CCW in the channel program.

Normally, an incorrect length condition is considered an error. For example, this would arise if the count field of a CCW did not agree with the actual number of bytes in a magnetic tape record (determined by the number of bytes recorded between two interrecord gaps). Sometimes it might be desired, however, to read only the beginning of a long record. When SLI is 1, the associated error condition interruption is suppressed. When data chaining is specified, of course, "correct" lengths are required, and incorrect length indications cannot be suppressed.

When SKIP is zero, normal transfer of information to memory during Read, Read Backward, and Sense occurs. When SKIP is 1, this transfer is suppressed. This feature is useful for selectively reading some portions of input records and for skipping others.

When PCI is zero, the straightforward flow of a channel program from CCW to CCW takes place, but when PCI is 1, an interruption is generated

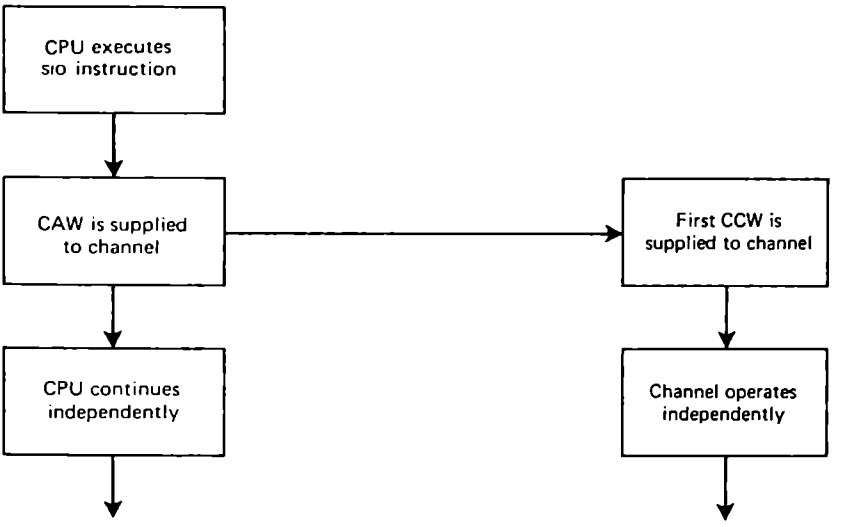


Figure 10.1 Initiation of input/output operation

as each CCW is fetched from memory. This feature provides the ability to monitor the operation of a channel program in order to keep ongoing processing attuned to current operating conditions.

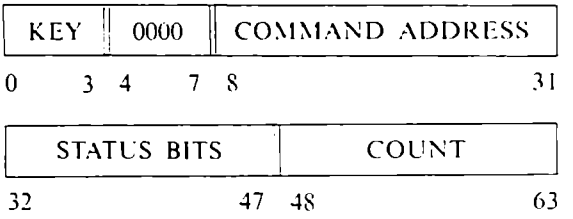
It was stated above that the execution of the SIO instruction initiates a chain of events, the first of which is the movement of the CAW from a fixed position in memory to the channel specified in the first (and only) operand. The second is the movement of the first CCW, under the control of the channel, from memory to the channel. Subsequent events are dependent upon the execution of the channel program and the flags specified in the commands. The manner in which the CPU communicates with a channel to initiate an input/output operation is summarized by the flowchart of Figure 10.1.

Especially during multiprogramming operations, it is to be expected that the above sequence might not always be possible. A channel or a device addressed by one program might be in use by another; an error condition might be encountered, and so on. It is necessary to have a way to determine what has in fact occurred when an input/output operation has been initiated, and this is the function of the Channel Status Word.

10.1.3 The Channel Status Word

The number of conditions that may arise during execution of SIO and the other input/output instructions is such that the two-bit condition code cannot contain the necessary status information. System/360 uses (and reserves) the doubleword at locations 64-71 (called the Channel Status Word) for this purpose. The contents of the CSW are automatically formed

during the execution of input/output instructions and during the course of input/output interruptions, in much the same way that the condition code is formed during the execution of instructions generally. Also like the condition code, the CSW resulting from a particular event is available to the programmer for testing and other use until a subsequent input/output event causes an alteration. The format of the CSW is as follows:



10.1.3.1 CSW Command Address

The first half of the CSW is the same as the CAW except for the interpretation of the command address. Whereas the command address portion of a CAW refers to the first command of a channel program, that of a CSW contains the address of the CCW following the last CCW used. It is analogous, therefore, to the instruction address portion of a PSW.

10.1.3.2 CSW Count Field

The count field of a CCW contains the number of bytes to be transferred during the course of the associated input/output operation. During execution, the channel decrements this field by one for each byte transferred. If the operation goes through to completion successfully, the count field is zero at that time; if for some reason the operation is terminated before completion, the count field at such termination contains the number of bytes not yet transferred. Whichever is the case, the residual count field of a CCW becomes the count field of the CSW and is placed in bit-positions 48–63.

Each of the sixteen status bits in positions 32–47 of the CSW specifies by a 0 or a 1 whether a given condition associated with the input/output operation is absent or present. These conditions are listed in Table 10.1 and discussed below.

The sixteen status bits of the CSW are divided into two groups of eight each. The first eight are all associated with control units and input/output devices; the last eight are all associated with channels. Between a channel and the control units attached to it are the so-called *standard input/output interfaces* (Sec. 1.2.1). What this means is that System/360 has a standard channel design to which input/output devices are interfaced in a standard

TABLE 10.1 INTERPRETATIONS OF CSW STATUS BITS

Bit Position	Condition
32	Attention
33	Status Modifier
34	Control Unit End
35	Busy
36	Channel End
37	Device End
38	Unit Check
39	Unit Exception
40	Program-Controlled Interruption
41	Incorrect Length
42	Program Check
43	Protection Check
44	Channel Data Check
45	Channel Control Check
46	Interface Control Check
47	Chaining Check

way. It is the function of the control units to match the characteristics of particular devices to the standard interfaces.

1. Certain input/output devices require synchronization for proper operation. The Attention bit records an asynchronous condition significant to the program. If such a condition is sensed prior to the input/output operation taking place, the operation is suppressed; otherwise, command chaining is suppressed.
2. In certain situations, such as those having a pending interruption condition at an input/output device, it may become impossible to execute a command as specified in the channel program. In such cases, the Status Modifier bit is generated to provide the necessary information to the program.
3. An I/O control unit may normally be considered part of the device and not a separate entity. In certain cases, however, such as with magnetic tape drives, several devices ordinarily share a single control unit, and it is important to be able to recognize when a shared control unit has become available for use with another of its devices. The Control Unit End bit provides this information. It is generated only for shared control units and only if the condition of the control unit has been interrogated by CPU instructions (discussed below).
4. The Busy bit is generated when a command or instruction cannot be executed because the input/output device or control unit specified is already in operation or because an interruption for either the device or the unit is pending.

5-6. The Channel End bit is generated at the end of each channel program and indicates that the channel is available for execution of

another program; the Device End bit is generated at the end of each input/output operation and indicates that the device is available for execution of further operations. In some cases the Channel End and Device End bits are generated at the same time and sometimes not, depending upon the type of device and the type of input/output operation involved.

7-8. An error condition, whether the result of programming or of equipment malfunctioning, that makes it impossible for an input/output device or a control unit to operate properly is signalled by the Unit Check bit. Unit Exception, on the other hand, does not necessarily signify an error condition. Its function is to indicate an exceptional condition, the exact meaning of the indication being different for each type of input/output equipment. For example, Unit Exception is used in conjunction with magnetic tape operations to note the sensing of a tape mark.

9. The Program-Controlled Interruption bit, when set on, indicates the presence of a PCI Flag in the CCW fetched by the channel. Thus the special processing for which the PCI Flag was placed in the CCW can be made contingent upon the detection of this bit in the CSW.

10. The Incorrect Length bit is set on when there is a mismatch between the CCW count field and the number of bytes actually transferred. As previously discussed, this condition can be suppressed by use of the SLI Flag.

11-12. The Program Check bit is generated when an error in the channel program is detected, and the Protection Check bit is generated when the storage-protection key associated with an input operation does not match the key assigned to the memory input area. The latter check applies only when the System/360 has the protection feature installed and when neither of the keys is zero.

13-15. The Channel Data Check bit is set on when data being transferred to or from an input/output device are detected in memory or in the channel with an even number of bits in a byte. Such detection is performed with the parity bit, the ninth bit in each byte, which is set such that during correct operation the number of 1's in each byte is odd. The Channel Control Check bit is set on when incorrect parity in a CCW address or in the contents of a CCW field is detected or when machine malfunctioning affects channel controls. The Interface Control Check bit is set on when the same sorts of errors are detected at the input/output interface.

16. In some cases it is possible for an input data rate that is very high to cause a Chaining Check bit to be set on during a data chaining operation. This occurs when the particular sequence of data addresses is such that chaining is not accomplished fast enough to keep up with the device.

10.2
I O AND RELATED INSTRUCTIONS

In addition to the SIO instruction mentioned previously, System 360 provides several basic instructions associated with input output processes. We shall include SIO in this discussion to preserve continuity.

10.2.1
The SIO Instruction

This privileged SI instruction has the general form

SIO *operand*

where *operand* provides the channel program with information which identifies the channel and the device which will be involved in the I O operation. This must eventually be expressed as an eleven-bit binary number in which the first three define the channel and the other eight identify the particular device. The channel designation may be 000 (which represents the multiplexer channel) or 001 through 110 (which respectively represent one of the six selector channels). The device designation may be anything from 0 to 11111111 (255_{10}) and will depend on how the devices are connected in a particular installation.

In many cases it is convenient to place this information in a general purpose register. This is done indirectly, by loading a register with the *address* of an area in storage containing device information. When the SIO instruction is used in this way it takes the form

name SIO 0(*n*)

where *n* specifies the register number containing the address of the place in which the device is defined.

The Condition code is set as follows:

Condition Code	*Condition*
00	Input/output operation initiated and channel proceeding with its execution.
01	Execution cannot proceed for reasons other than those for which the condition code is 10 or 11; CSW is stored.
10	Channel or subchannel busy.
11	Not operational.

Condition code 10 is set if the channel or subchannel is busy. A *subchannel* is that part of a channel's circuitry required to control a single input/output operation, including storage for addresses, count, and status and control information. Each selector channel has a single subchannel and can control only one input/output operation at a time. A multiplexer channel has multiple subchannels (as described in Sec. 1.2.1.3).

10.2.2 THE ccw Instruction

This assembler instruction allows the program to fill a designated doubleword with information that will be used as a Channel Command Word. The contents are appropriately converted and automatically placed at a doubleword boundary. Four operands are required as follows:

1. A constant specifying the command to be executed. The repertoire of channel commands is:

1. Write	4. Sense
2. Read	8. Transfer In Channel
3. Control	12. Read Backward

   These codes may change from installation to installation.
2. The address involved in the operation. This is generally the memory location from or to which data are to be transmitted.
3. A hexadecimal constant that will be used to set the five CCW flags (Sec. 10.1.2.3) and the next three bits to zero.
4. A constant specifying the number of bytes to be transmitted. For example, if we wanted to write a 120-byte field on a printer and the first byte had a symbolic address of LINE, our ccw could be written as

   ```
   TELLIT   ccw     1,LINE,x'00',120
   ```

   In this particular case, the name TELLIT will be assigned to the first byte of the CCW.

EXAMPLE 47

Write a sequence of instructions that will start a process whereby all 80 columns of a card will be read into a field named HERE. The card reader is attached to the multiplexer channel and its assigned address is 12.

Solution Procedure

1. Set up a CAW and place it in location 72. (Call its original location CHAD).
2. Set up a CCW (we shall assign the name COMWD to it).
3. Set up a word designating the card reader (multiplexer channel, device 12) and store its address in a general register (we shall use Register 5 and the name WHICH).
4. Start I/O.

Discussion

Since our CAW must refer to the address of the first CCW for the operation (the only CCW in this case), its operand will be COMWD. The CCW will call for the channel to Read (Code 2) into memory (HERE) with all flags set to zero (x'00') and will specify that 80 bytes are to be transmitted.

Solution Program

```
Name      Operation   Operand          Comment
          START       5000
BEGIN     BALR        14,0
          USING       *,14
          L           7,CHAD           CAW IN REGISTER 7
          ST          7,72             CAW in 72
          LA          5,WHICH
          SIO         0(5)

          .           .
          Other instructions (not pertinent here)
          .
          DS          0F
CHAD      DC          X'0000000C'
WHICH     DC          X'00',A(COMWD)   COMWD'S ADDRESS IS WHICH
HERE      DC          CL80
COMWD     CCW         2,HERE,X'00',80
          END         BEGIN
```

Since SIO is a privileged instruction, the activities triggered by it (such as fetching the CAW from location 72 and subsequent accessing of the CCW) must be preceded by a status switch from program to supervisor state. After the supervisory package executes the SIO and turns control over to the appropriate channel, another switch occurs and the processor can continue with ensuing instructions while the input/output is going on. Example 47 is included to illustrate the basic sequence involved in starting this activity. However, it does not represent the way such operations are handled in most problem programs.

10.2.3 The Halt I/O Instruction

An I/O operation is terminated by the privileged SI instruction

name HIO *operand*

where *operand* has the same meaning as in the SIO instruction (10.2.1). The condition code is set as follows:

Condition Code	*Condition*
00	Interruption pending in subchannel
01	CSW stored
10	Burst operation terminated
11	Not operational

In a situation such as the one presented in Example 47, the HIO would follow the SIO instruction, closing out that particular I/O operation.

10.2.4 The Test I/O Instruction

This instruction has the same form as SIO and HIO (with TIO as its mnemonic code). Its purpose is to determine the status of the device and channel specified in its operand. The condition code is set as follows:

Condition Code	*Condition*
00	Available.
01	Not available for reasons other than those for which the condition code is 10 or 11; CSW stored.
10	Channel or subchannel busy.
11	Not operational.

10.2.5 The Test Channel Instruction

This instruction is also privileged and has the form

name TCH *operand*

where *operand* usually refers to a storage address. The channel specified by bits 16-23 of that operand is examined and the condition code is set as follows:

Condition Code	*Condition*
00	Channel available.
01	Interruption pending in channel.
10	Channel operating in burst mode.
11	Channel not operational.

10.2.6 Additional Instructions

A number of additional instructions are provided by special features. These are not supported by the basic assembler and are listed here for reference. Their use is generally restricted to supervisory programs.

10.2.6.1 Read Direct

This SI instruction RDD is used to obtain input from another CPU rather than a peripheral device. When it is specified, it causes a byte to be read from an attached CPU. The byte is stored in the address given by the first operand; the immediate second operand is used as a timing signal.

10.2.6.2 Write Direct

This, like RDD, is also a privileged SI instruction having the mnemonic code WRD. Its execution transmits the byte in the first operand address to another CPU.

10.2.6.3 Set Storage Key, Insert Storage Key

These privileged RR instructions (mnemonic codes SSK and ISK) are used to establish protection status for designated 2,048-byte blocks of memory. The protection feature must be installed for these to operate.

10.2.7 Preparations for Interrupt Handling

Several additional privileged instructions are available to the supervisor for handling interrupts, such as those which must precede and follow the execution of I/O operations. As before, these are listed merely for reference, since the tools available to the applications programmer automatically activate supervisory procedures which include these types of instructions.

10.2.7.1 Set Program Mask

Although this is an RR instruction (mnemonic SPM), it has only one operand. Bits 2–7 of the register specified in that operand are reproduced in bits 34–39 of the current PSW, thus becoming the new condition code and program mask fields. Note that it is these bits that are obtained from the PSW by the BAL and BALR instructions (Sec. 6.4.2).

10.2.7.2 Set System Mask

The SSM instruction changes the system mask bits (bits 0–7) of the current PSW. It is an SI instruction with only one operand. The byte in the storage location specified in the operand becomes the system mask.

10.2.7.3 Test and Set

This SI instruction, with mnemonic TS, is in the same form as SSM (SI type with one operand). When it is executed, bit 34 of the PSW is set to 0, bit 35 is set to the value of the first bit in the operand, and then the entire operand byte is set to 1's. Thus the condition code reflects the value of the leftmost bit of the operand.

10.2.7.4 Load PSW

This privileged instruction, LPSW, is also of the SI type but has a single operand, like SSM. That operand must refer to the first byte of a doubleword whose contents become the current PSW, with the PSW's previous contents being lost.

10.2.7.5 The Supervisor Call Instruction

Although SVC is an RR instruction, it takes a single operand that is usually expressed as a decimal constant from 0–255 (or x'00' to x'FF'). When an SVC instruction is encountered, a supervisor-call interruption occurs. As a result, the old PSW is stored at location 32 and the new PSW is obtained from location 96. The byte specified in the SVC operand is stored in bits 24–31 of the old PSW and therefore constitutes a code indicating the reason for the machine status change. Bits 16–23 of the old PSW are set to zero as part of the process.

Thus SVC is the means by which the problem programmer is able to switch the CPU from the problem state to the supervisor state. Although he does not write the privileged instructions and others that are used for I/O, these are part of the supervisory program and they call on information supplied by the problem program. For example, if the programmer wants to start an I/O operation, he will specify the appropriate code in the SVC operand or in some less direct way. The supervisor will use that code to determine the address to which it will branch in order to execute its "start I/O" routine.

10.3 MACRO INSTRUCTIONS

For simplicity of programming, the problem programmer does not use SVC directly and does not create his own channel programs. Instead, he communicates with the supervisory program by means of *macro instructions*. These are preprogrammed assemblages of machine instructions and assembler instructions to which the problem programmer has access by means of single statements. One of the simpler macro instructions—used in all the previous examples—is EOJ (End of Job). This is translated by the assembler into a Supervisor Call instruction with an immediate operand code that is interpreted to mean that the problem program execution is complete. In this way it is possible for individual programs to be terminated without bringing the system to a halt. Similarly, the CCW statement is actually a macro instruction for defining a Channel Command Word, even though the programmer uses it as a single instruction.

Macro instructions have the appearance of machine and assembler in-

structions. They may have name fields; they always have operation fields; and they have operand fields with various numbers of operands. The formats of macro instructions, like those of assembler instructions, do not conform to the five standard formats of machine instructions but are designed according to the functions performed.

For some versions of Assembler Language, there are both basic and advanced input/output macro instructions. Following below is a set of basic I/O macro instructions available to virtually all System/360 configurations.

10.3.1 The EXCP Macro

If the programmer wishes to initiate some I/O operation he calls for the supervisor to activate a particular channel program. This is triggered by the Execute Channel Program macro, whose form is

name EXCP *blockname*

The single operand designates the symbolic address of a CCB macro instruction.

10.3.2 The CCB Macro

The Command Control Block (CCB) macro, referenced by an EXCP, has two required operands and a third optional one. The general form is

blockname CCB SYS*nnn*,*ccwname*,X'*yy*'

The first operand is the symbolic name of the input/output device to be used. Allowed names are SYSRDR (system reader), SYSLST (system lister), SYSIPT (system input), SYSOPT (system output), and SYS000–SYS255 (system devices 000–255). Assignment of actual input/output devices (the correspondence between device numbers and physical units) is ordinarily made via a job control card by the system operator, not by the problem programmer.

The second operand is the symbolic name of the first command in the channel program, in other words, the first CCW.

The execution of CCB causes an eight-byte field to be created in memory. In this field are stored data used by the supervisory program in carrying out input/output operations, and in it also are stored status bits after operations for use by the problem programmer. The optional third operand of CCB is a hexadecimal constant that refers to bytes 2 and 3 of the eight-byte block and may be used to set any of sixteen bits (1 = on, 0 = off). Ordinarily only the last five bits of byte 2 are used by the programmer in this manner, but all are accessible. The functions of all sixteen of these bits are given in Table 10.2

TABLE 10.2 CCB BYTES 2 AND 3

Byte 2	
Position	**Function when bit value = 1**
0	I/O Completed (Channel End)
1	Bit 41 in CSW Indicates Wrong-Length Record
2	Unrecoverable I/O Error
3	Accept Unrecoverable I/O Error
4	Indicates 1015 (Special Device) CCB
5	Test for Punch Errors or Carriage Channel 9 or 12 Overflow
6	Accept Tape Read Error
7	User Error Routine

Byte 3	
Position	**Function when bit value = 1**
0	Not Used
1	Not Used
2	Not Used
3	2540 Equipment Check or Tape Read Error
4	Unusual Command Sequence or Other Questionable Condition
5	Carriage Channel 12 Overflow or End of File
6	Carriage Channel 9 Overflow
7	Device End for Printer Occurred

10.3.3 The WAIT Macro

The form for the macro is

name WAIT *blockname*

where *blockname* has the same meaning as in EXCP and CCB. WAIT is used whenever the program requires that an I/O operation started by an EXCP instruction be completed before execution of the problem program continues. These, together with CCW (Sec. 10.2.2), constitute the four basic macro instructions available to the programmer for I/O.

He uses the EXCP/WAIT pair wherever he wishes to execute a particular operation, and he uses a CCB and one or more CCW's to define and specify the supporting information. EXCP, especially, saves considerable time and effort for the problem programmer by generating the SVC instruction with its subsequent interruption and by handling all aspects of the SIO instruction.

EXAMPLE 48

Each card in a deck of 2,000 cards has punched in it factors A, B, and C. These factors are punched in columns 1–8, 9–16, and 17–24, respectively, and are right-justified decimal integers punched in the standard card code. Write a program to reproduce this deck in such a way that A, B, and C are in the same positions, the

sum of A, B, and C is in columns 25–40, and a sequence number for each card is in columns 77–80. Also, list the output cards on the printer, one line per card.

Solution Procedure

1. Set up an iterative loop to count the cards.
2. Read a card.
3. Update the sequence number.
4. Convert A, B, and C to packed-decimal format and form their sum.
5. Convert the sum and the sequence number to zoned-decimal format.
6. Punch a card and print a line.
7. Continue until the deck has been reproduced.

Solution Program

```
Name      Operation   Operand
          START       10000
IO        BALR        15,0
          USING       *,15
          SP          NUMBER,NUMBER
LOOP      EXCP        CARDIN
          WAIT        CARDIN
          AP          NUMBER,ONE
          PACK        FIELD,INOUT(8)
          ZAP         SUM,FIELD
          PACK        FIELD,INOUT+8(8)
          AP          SUM,FIELD
          PACK        FIELD,INOUT+16(8)
          AP          SUM,FIELD
          UNPK        INOUT+24(16),SUM
          UNPK        INOUT+76(4),NUMBER
          EXCP        CDOUT
          WAIT        CDOUT
          EXCP        LIST
          WAIT        LIST
          CP          NUMBER,TWOTHO
          BC          4,LOOP
          EOJ
NUMBER    DC          XL3'C'
CARDIN    CCB         SYSRDR,RDCCW
ONE       DC          X'1C'
FIELD     DS          CL5
INOUT     DS          CL80
SUM       DS          CL5
CDOUT     CCB         SYSOPT,PCHCCW
LIST      CCB         SYSLST,PRTCCW
TWOTHO    DC          XL3'2000C'
RDCCW     CCW         2,INOUT,X'0',80
PCHCCW    CCW         1,INOUT,X'0',80
PRTCCW    CCW         1,INOUT,X'0',80
          END         IO
```

Discussion

This program is intended to show how the four macro instructions introduced above are incorporated into a program. Both the usage of the macro instructions and the overall requirements of the example are kept simple to attain this objective. Despite this approach, however, it has been possible to illustrate in a single short program how cards may be read and punched and how printed output may be obtained.

The use of EXCP and WAIT in pairs and, in general, with no intervening statements, raises a question about buffering. If an input/output operation is initiated, and if the processor is made to wait until I/O has been completed before further processing can continue, what is the advantage of having a buffer in an input/output device? Referring to the card-reading operation in answering this question, it should be realized that when the command to read the next card is issued, the next card is already in the buffer (unless the program is input-bound) and that the operation consists merely of transferring its contents to memory; the mechanical motion of reading the following card takes place after this transfer. The advantage of buffering is thereby preserved through use of these macro instructions.

Ordinarily, more attention must be paid to printed output than was given in this Solution Program. Fields are normally edited using the techniques already discussed, they are spaced to provide readable and attractive formats, alphabetic constants are inserted for identification purposes, and so on.

Headings and page numbers are often desired on each page for output reports consisting of multiple pages. The printer contains a paper-tape-controlled carriage with holes punched in various tape channels. The positions of these holes, among other uses, regulate at what point printing on one page is terminated and at what point on the next page printing is begun. In order for the problem programmer to be able to insert headings and page numbers properly, he must be able to test the associated positions of the printer carriage. This is accomplished in conjunction with the CCB. Referring to Table 10.2, note that bit position 5 of byte 2 gives him the ability to specify that carriage channels be tested by the supervisory program and that bit positions 5 and 6 of byte 3 contain the result of such tests. He uses the Test Under Mask (TM) instruction after each line is printed to determine from these bits the position of the page with respect to the printer carriage.

The use of SYSRDR, SYSOPT, and SYSLST as the symbolic names of the card reader, card punch, and printer, respectively, is in line with the practice at many installations but is not necessarily standard. Similarly, the assignments of SYSIPT and SYS000-SYS255 at various installations, while not actually random, are primarily at the option of each installation. Each problem programmer, then, should familiarize himself with the items that are specific to his own location. Among these are the symbolic names of the input/output devices and also the possible modifications to the first operand of CCW's, the number of characters that can be printed per line, and the standard job control cards used.

PROBLEMS

1. Why is input/output processing more difficult than internal processing? What determines how rapidly a particular program can be processed?

2. What are privileged instructions and why are they used?
3. What function do macro instructions play? What are the differences in Assembler Language between machine instructions, assembler instructions, and macro instructions?
4. How is it possible for the problem programmer to execute Start I/O by using EXCP when he is unable to execute it directly?
5. In what sort of situations might channel programs consist of more than one command?

In each of the following exercises the deck of 2,000 cards of the example is used. Each of these cards has integers A, B, and C punched in columns 1–8, 9–16, and 17–24, respectively.

6. Write a program to reproduce the card deck with A, B, and C in columns 1–8, 21–28, and 51–58, respectively.
7. Write a program to produce a listing of the cards, one line per card. Identify each field using its symbolic name and an equals sign as follows:

```
A = XXXXXXXX
B = XXXXXXXX
C = XXXXXXXX
```

8. Repeat the previous exercise, also editing the fields so as to remove leading zeros, to insert commas as needed, and to precede each negative quantity with a minus sign.
9. Repeat Problem 7 with a blank line between each pair of information lines. Also, write the program as if the number of cards is unknown: that is, keep reading cards and printing lines until the unit-exception bit (sixth bit of the third byte in the CCB) is found to be one. [*Hint:* use the TM instruction to make this determination.]
10. Write a program to store the information in the deck of cards on magnetic tape. Group each 50 cards into one magnetic tape record, first converting each value to a binary word. (Each tape record contains 50 cards times three words per card times four bytes per word, or 600 bytes.) Use SYS001 as the symbolic name of the magnetic tape unit.
11. Write a subroutine that converts an eight-digit zoned-decimal integer into long-operand floating-point format. [*Hint:* After converting to binary format, each group of four bits, counting from the right, is a binary-coded-hexadecimal digit. After constructing the floating-point doubleword, normalization may be accomplished automatically by adding it to a floating-point zero.]
12. Repeat Problem 10 using the subroutine of Problem 11 to convert each field to long-operand floating-point format prior to storage on magnetic tape.
13. Reproduce the original deck of cards such that each three cards of the original deck are punched into a single card of the new deck. (Note that the last card of the new deck is punched in columns 1–48, not in columns 1–72 like the others.)

14. Reproduce the original deck of cards such that all columns of the new deck are punched. (The first card of the new deck contains the first three cards of the original deck plus the first field of the fourth; the second contains the second and third fields of the fourth card, all of the fifth and sixth cards, and the first and second fields of the seventh card; and so on.)
15. Print a report using the original deck of cards as the source and following these directions:
    (a) Print the column headings: A, B, C, A+B, B-C such that they are centered over positions 1-12, 21-32, 41-52, 61-72, and 81-92. Then skip a line.
    (b) Under these column headings print a line for each card, making the appropriate conversions and computations.
    (c) At the end, skip two lines and print the sum of each column; then skip two more lines and print "END OF REPORT".

FURTHER READING

1. *Basic Operating System—Assembler with Input/Output Macros*, Form C24-3361. IBM Corp., White Plains, N.Y.
2. *Tape/Disk Operating System—Assembler Language Specifications*, Form C24-3414. IBM Corp., White Plains, N.Y.
3. *Operating System—Assembler Language Specifications*, Form C28-6514. IBM Corp., White Plains, N.Y.
4. *Operating System—Assembler Language, Programmer's Guide*, Forms C28-6595, C28-3756. IBM Corp., White Plains, N.Y.

Chapter 11

Additional Assembler Language Features

This chapter contains a partial catalog of assembler and macro instructions available in one or more versions of Assembler Language. Just as with optional input/output devices, the programmer must verify which are installed at his particular location in order to make optimal use of this language. A complete catalog of assembler and macro instructions available in each particular version of Assembler Language is not included because it is an inadequate guide to what is available at a given location. The reason is that Assembler Language is an open-ended language to which programmers may make additions. Such additions are in the form of macro instructions which look no different to the problem programmer from the assembler and the macro instructions presented here.

In Chapter 4, a list of basic Assembler Language assembler instructions is given. Some of these—DC, DS, CCB, CCW, START, USING, END—have already been used and described. The catalog in this chapter continues with additional assembler instructions, and then ends with advanced macro instructions. For each such instruction, the name given is the operation field and the discussion describes the operands. Those operations described in detail and used previously will not be reconsidered here.

11.1 ASSEMBLER INSTRUCTIONS

11.1.1 The EQU Instruction

EQU (for Equate Symbol) is used to define a symbol by assigning to it the value and the length attributes of an expression in the operand field. It is the usual way to equate programmer-designated symbols to register numbers, input/output unit numbers, immediate data, actual addresses, and compound expressions, as illustrated in the following examples:

```
Name    Operation   Operand
REG2    EQU         2
I0125   EQU         125
TEST    EQU         X'3F'
TIMES   EQU         80
FIELD   EQU         A-B-C
```

11.1.2 The START Instruction

START is normally the first instruction in a program unless ICTL (see below) is used. A symbol in its name field is identified as the program name. A constant in its operand field is taken to be the initial setting of the location counter.

11.1.3 The ORG Instruction

ORG (for Origin) contains in its operand field an expression that resets the location counter to a relative value.

```
Name    Operation   Operand
        ORG         *+500
```

In this example, note that the name field is not used and that the location counter is set to a value 500 bytes above the value when the ORG instruction is encountered. Nothing is assembled for the 500 bytes skipped; they are not cleared by the assembler; and they should not be assumed to be set to zero or anything else.

11.1.4 The ICTL Instruction

Source program statements are assumed by the various System/360 assemblers to be punched in certain card columns. ICTL (Input Control) may be used to alter such ground rules, but its use is a function of the

assembler installed. In the most basic assembler, for example, the operand field may contain either 1 or 25, specifying the card column in which source statements begin. In an advanced assembler, on the other hand, the operand field may contain three constants separated by commas. The first (from 1–40) specifies the card column in which source statements begin; the second (41–80) specifies the last card column in which source statements may appear; and the third (2–40) specifies the card column in which continuations of statements begin. When used, the ICTL statement must precede all other statements in the source program and may be used only once.

11.1.5 The CNOP Instruction

CNOP (Conditional No Operation) allows the programmer to align an instruction at a specific halfword, word, or doubleword boundary, analogous to the use of the zero multiplier in a DS operand to produce the same result for data fields. In addition, whenever such alignment causes the location counter to be advanced, the intervening halfwords are filled with BC (Branch on Condition) instructions with zero operands. Possible CNOP operand fields, with resulting alignment, are as follows:

Operand Field	*Alignment*
0,4	Beginning of a word
2,4	Middle of a word
0,8	Beginning of a doubleword
2,8	Second halfword of a doubleword
4,8	Third halfword of a doubleword
6,8	Fourth halfword of a doubleword

This instruction is useful in creating subroutine linkages followed by parameters which require boundary alignment.

11.1.6 The EJECT Instruction

EJECT, with the name and operand fields left blank, is inserted in the source program wherever it is desired to continue the assembly-time program listing at the beginning of a new page. This provides a convenient way to produce a program listing organized into logical segments. The EJECT statement itself is not printed; if this is desired, the name and/or the operand field is made nonblank.

11.1.7 The SPACE Instruction

SPACE is used to insert one or more blank lines in the assembly-time program listing, a decimal constant in the operand field specifying the number of blank lines. A blank operand field causes one line to be skipped.

If the number of blank lines to be inserted exceeds the number of lines remaining on the listing page, SPACE has the same effect as EJECT. The SPACE statement itself is omitted from the listing when the name field is blank, otherwise it is included.

11.1.8 The USING Instruction

USING indicates to the assembler that the general register specified as the second operand is available for use as a base register and that the contents of this register are assumed to be as designated by the relative address in the first operand field. Multiple USING statements, appearing anywhere in a source program, may be used. The following example illustrates usage beyond that shown earlier:

```
Name    Operation   Operand

        .
        USING       *,11
        USING       *+4096,12
        USING       *+8192,13
        USING       *+4500,14
```

The assembler enters the information given to it in USING statements into a *register table*. Whenever an effective address is specified in a machine instruction statement, the assembler consults this table to determine whether there is an available register containing a suitable base address. If more than one register produces a valid displacement, the register whose contents produce the smallest displacement is used. In the case of a tie, the higher number register is used.

Because the range of a displacement is 4,096 bytes, some versions of Assembler Language enable multiple specification of base registers with 4,096-byte increments by means of a single USING statement. The following statement is of this kind and is equivalent to the first three USING statements above:

```
Name    Operation   Operand

        .           .
        USING       *,11,12,13
```

11.1.9
The DROP Instruction

DROP, whose operand field contains an expression (normally just a decimal constant), specifies to the assembler a previously available register that may no longer be used as a base register. DROP reverses the effect of USING, therefore, and removes the specified entries from the register table.

11.1.10
The ENTRY Instruction

ENTRY provides a means to identify a symbol defined in the same program that is to be used as an entry point to that program from another program or other programs. In some versions of Assembler Language, this statement must follow START or another ENTRY statement and may contain only one symbol in the operand field; in other versions, there is no restriction on placement of the statement or number of symbols in the operand field. Multiple symbols, when used, are separated by commas.

11.1.11
The EXTRN Instruction

EXTRN (External Link) identifies a symbol defined in and used for branching to another program; in the latter program, it is an entry point and is identified using an ENTRY statement. In some versions of Assembler Language, this statement must follow either a START, ENTRY, or another EXTRN statement and may have only one symbol in its operand field; in other versions, there is no restriction on placement of the statement or number of symbols in the operand field. A v-type address constant defined in a DC statement does not have to be identified in an EXTRN statement.

11.2
ADVANCED ASSEMBLER INSTRUCTIONS

11.2.1
The CSECT Instruction

Large programs are often written in sections to facilitate production by more than one programmer and to permit modular maintenance. Assembler Language aids this approach by enabling the definition of *control sections*. When a program is named by means of a START statement, that name identifies what is considered the first (it may be the only) control section in the program. When it is desired to identify further control sections, CSECT (Identify Control Section) is used. CSECT normally has a symbol in its name field and is always blank in its operand field. It may be used anywhere in a program and identifies either the beginning or the continuation of a control

section. Several CSECT statements with the same name may appear within a program, the first identifying the beginning of the control section and the remainder identifying its resumption. A control section continues until a CSECT statement, a DSECT statement (see below), or the end of the program is reached.

There may be one unnamed control section in a program. Either START or CSECT may appear without a name, therefore, to signal the beginning of this control section, and CSECT may also appear without a name to signal its continuation.

11.2.2 The DSECT Instruction

DSECT (Identify Dummy Section) always has a symbol in its name field and a blank operand field. It is used to identify control sections of a program that are assembled but are not made part of the object program.

11.2.3 The COM Instruction

COM (Define Blank Common Control Section) is a statement with blank name and operand fields that reserves an area of memory for common reference by independently assembled control sections linked and loaded for execution as a single program. The amount of memory reserved, which may be subdivided using DS and DC statements with names defined relative to the beginning of the common section, is equal to the longest common control section of those included in the assembly.

11.2.4 The TITLE Instruction

TITLE is a statement used for identification of the assembly listing and of the assembly output cards. The operand field consists of a sequence of up to 100 characters enclosed in single quotation marks; this sequence is printed at the top of each page of the assembly listing. If it is desired to include either a quotation mark or an ampersand as part of this title, two of such characters are inserted. TITLE statements after the first replace the previously used page title and are used until replaced by a subsequent TITLE statement.

The name field may consist of from one to four alphabetic and/or numeric characters. When this field is so used, the assembly output cards are identified with this name in columns 77-80. If there is more than one TITLE statement in a program, only the first may have a nonblank name field. If PUNCH or REPRO statements (see below) are used, they take precedence over the punching of assembly output cards.

11.2.5 The PRINT Instruction

PRINT is a statement used to control the printing of the assembly listing. It has a blank name field and from one to three operands in the operand field, separated, if need be, by commas. The first operand is either ON or OFF, specifying that an assembly listing is or is not to be printed, respectively. The second is either GEN or NOGEN, specifying that all statements generated by macro instructions are or are not to be printed, respectively. The third operand is DATA or NODATA, specifying that constants are printed in full in the listing or are ignored except for their leftmost eight bytes. The operands of PRINT form a hierarchy. If the first is OFF, the second and third are irrelevant; if the first is ON, then those that follow are considered. If the first is ON and the second is NOGEN, the third is irrelevant with respect to macro instructions; if the first is ON and the second is GEN, then the third controls the printing of all constants.

11.2.6 The ISEQ Instruction

ISEQ (Input Sequence Checking) is used to check the sequence of source language cards during assembly. Its operand field is used to specify the card columns to be checked in the form a,b. If this field is 72,75 and these columns have not been defined by ICTL as a part of the statement area, for example, each card following the ISEQ statement is checked to ensure that the contents of card columns 72–75 are greater than the contents of those columns in the previous card. The comparison is made after each character in the specified columns is converted to its eight-bit internal equivalent.

If groups of input cards have been punched independently and have different formats and/or sequence numbering, the checking of one group can be discontinued by inserting an ISEQ statement with a blank operand field, and the checking of a subsequent group can be initiated with an ISEQ statement including new format specifications.

11.2.7 The PUNCH Instruction

PUNCH is used to punch a card of the assembly output deck. One punch statement causes the punching of one card, and as many such statements may be used as desired. The name field is blank, and the operand field is used to specify the information to be punched. The latter field contains a sequence of from one to eighty characters enclosed in single quotation marks. As with TITLE, quotation marks and ampersands within

the desired sequence must be represented by a pair of such characters. The first character of the sequence is punched in the first column of the associated card.

11.2.8 The REPRO Instruction

REPRO is used to reproduce the source deck card that follows the statement into the assembly output deck on a column-for-column basis. Both the name and operand fields of this statement are blank.

11.2.9 The LTORG Instruction

Constants have normally been specified as self-defining terms within instructions or in DC statements referenced symbolically. Another way to introduce a constant into a program is by means of a literal. Although we have discussed literals (Chapter 4), we have not used them thus far. For example, in the statement

```
Name   Operation   Operand
LOAD   L           10,=F'4'
```

the literal operand =F'4' is written as data although the quantity cannot be assembled into the instruction. In function, a literal is treated as if a reference to a constant defined elsewhere were being made; the constant is stored in a *literal pool* elsewhere in memory, and its address is assembled into the instruction.

LTORG (Begin Literal Pool) causes all literals since the previous LTORG statement (or since the start of the program) to be assembled at appropriate boundaries starting at the first doubleword boundary following the LTORG statement. The operand field is not used; the name field may be blank or may contain a symbol representing the address of the first byte in the literal pool. Only one literal is allowed in a machine instruction statement. If duplicate literals occur within the range controlled by one LTORG statement, only one literal is stored in the literal pool.

11.2.10 The COPY Instruction

This is used to obtain source-language coding from a library and include it in the program currently being assembled. The name field is blank, and the operand field contains a symbol that identifies the section of coding to be copied. Statements included in the program via COPY are assumed to be in standard coding format and are not subject to any ICTL statements to the contrary.

11.3 MACRO INSTRUCTIONS

Assembler Language contains macro instructions designed primarily to reduce the amount of repetitive coding required for general requirements that occur a number of times in the same or different programs. In addition, Assembler Language contains the capacity to be augmented by macro instructions constructed at individual installations to meet their own special requirements. The built-in macro instructions are very often used by problem programmers, especially in connection with input/output operations. Many of the macro instructions used both for these operations and for related definitions are described in the remainder of this chapter. A full description is not attempted, however, because of the strong installation-orientation of the subject.

11.3.1 File Definition

Numerous data processing applications are oriented toward the manipulation of information organized into files, and macro instructions facilitate both the definition and the treatment of such files.

Large files, such as might be stored on magnetic tapes or disks, ordinarily contain system-oriented information in addition to the application-oriented data. System-oriented information includes special records called *header labels* and *trailer labels* to facilitate system identification of each *volume*—that part of a file contained on a single reel of tape, disk pack, and so forth—and to expedite automatic volume-to-volume transitions. In addition, system-oriented information includes special indices used to access records in random (arbitrary) sequence, special symbols used to separate information blocks, etc.

The basic EXCP/WAIT approach to file processing has already been encountered (Sec. 10.3). A second approach is also provided in Assembler Language. For this method, the programmer must carefully define his files in considerable detail but is thereafter relieved of much of the programming, including the writing of channel programs, checking of file labels, and blocking and deblocking of records. Certain macro instructions that facilitate file definition are available for use when this approach to file processing is taken. Those that are mentioned are typical examples of macro instructions in use, rather than exhaustive examples of those available at a given location.

11.3.1.1 DTFSR

DTFSR (for Define The File in a Serial Device) is used to define files of logical records organized for consecutive processing. It has so many possible operands that a number of continuation cards must be used. Also,

a special keyword type of entry is used so that the operands may be entered in any order. A sample of the use of this macro instruction follows:

```
                                                                    Col. 72
                                                                      ↓
                                                                  Continuation
Name   Operation  Operand                                          Indicator

       DTFSR     TYPEFLE = INPUT, RECFORM = FIXBLK                    X
                 BLKSIZE = 400, RECSIZE = 80                          X
                 DEVICE = TAPE, READ = FORWARD,                       X
                 REWIND = UNLOAD, DEVADDR = SYS001,                   X
                 CONTROL = YES, FILABL = STD,                         X
                 IOREG = 3, IOAREA1 = AREAONE,                        X
                 IOAREA2 = AREATWO,LABADDR = CKOLDLAB,                X
                 ERROPT = CKOLDBLK, WLRERR = CKOLDWLR,                X
                 EOFADDR = EOFMSTR
```

Explaining the operands in the order in which they are shown, TYPEFLE specifies one of three types of file: INPUT, OUTPUT, or CMBND (combined). A CMBND file is one used for both input and output, such as a card file that is to be read and results punched in the same cards.

RECFORM specifies the format of the records in the file. They may be of fixed or of variable length, blocked, unblocked, or undefined. When several logical records make up a physical record, the records are said to be *blocked;* when a logical record is identical to a physical record, the records are said to be unblocked. RECFORM options are FIXUNB (fixed-length, unblocked), FIXBLK (fixed-length, blocked), VARUNB (variable-length, unblocked), VARBLK (variable-length, blocked), and UNDEF (undefined).

BLKSIZE specifies the number of bytes in the input or output area of memory. In the case of variable-length records, this area must be large enough to accommodate the largest block of records. In the case of blocked records, it must be large enough to accommodate an entire block.

RECSIZE specifies the number of bytes in each logical record. In this example, there are 80 bytes in a record and 400 bytes in a block; therefore, there are five logical records blocked into each physical record.

DEVICE options are TAPE, PRINTER, READ01 (for an input card file in a 2501), READ20 (for an input or output card file in a 2520), READ40 (for an input or output card file on a 2540), READ42 (for an input or output card file in a 1442), PTAPERD (for an input file on paper tape), and READ85 (for input from a 1285 optical reader), and CONSOLE, which speaks for itself.

For magnetic tapes, READ may be specified as FORWARD or BACK, according to the direction of motion when they are being read. Also for tapes, REWIND may be UNLOAD, to cause rewinding both before and after the associated files are processed, or NORWD, to prevent rewinding at any time.

DEVADDR (device address) specifies the input/output device by its symbolic name, one of SYSRDR, SYSLST, SYSIPT, SYSOPT, SYSLOG, or SYS000–SYS255.

The symbolic name is associated with an actual device by means of a system control card.

CONTROL = YES indicates that the CNTRL macro instruction will be used with this file. This macro instruction is used for card-stacker selection, carriage skipping, tape rewinding, etc. The programmer may want to handle these functions himself, in which case he uses the CTLCHR operand instead of CONTROL. When this is done, the CNTRL macro is not used. Instead, the first character of each record will be used by the system as a control indicator in accordance with a predefined code[1]. The programmer must make sure that the proper character appears in each record and its presence must be reflected in the value specified for BLKSIZE.

FILABL (file label) is STD (for a file with standard tape labels to be checked by the supervisory program), NSTD (for a file with nonstandard labels or standard labels that are not to be checked), or NO (for a tape file that does not contain labels.) If the NO option or NSTD option is specified and there is no tape mark (or, on output, no tape mark is required), the operand TPMARK = NO is included, otherwise "yes" is assumed.

IOREG specifies which one of the General Registers 2–11 is designated for use by the supervisory program's input/output routines. It is used for selecting which of a block of logical records is to be processed. Whenever macro instructions are used in a program, General Registers 0–1 are used by the generated instructions. General Registers 12–13 are ordinarily not used by the problem programmer either because of their use by the supervisory program in processing interruptions. In addition, General Registers 14–15 are used by the system in connection with the macro instructions GET, PUT, OPEN, and CLOSE (discussed below).

IOAREA1 and, optionally, IOAREA2 specify the symbolic names of the memory areas used when processing records. These areas must also be defined, and by the same names, in DS statements. The purpose of having two areas is to permit an overlap of data-transfer and data-processing operations by having one area act as a buffer for the other.

Instead of IOREG, the programmer may use the WORKA operand. This tells the system to allocate additional storage for receiving input or supplying output.

LABADDR is the symbolic address of a routine that the problem programmer may require to check tape labels in addition to the standard file header or trailer label. When such a routine is used, a special macro instruction called LBRET (label return) is used to return control to the supervisory program.

ERROPT has three options: IGNORE ignores input/output errors when sensed; SKIP goes on to the next block when such errors are sensed; or the specification of a symbolic address causes a branch to that location, usually for corrective action, when an input/output error is sensed.

TABLE 11.1 DTFSR OPERANDS FOR CARD READERS, CARD PUNCHES, AND LINE PRINTERS

Required Operand	Options	Optional Operand	Options
TYPEFLE =	INPUT OUTPUT	RECSIZE =	*n* (R)[b]
RECFORM =	FIXUNB VARUNB UNDEF	CONTROL =	YES
		CTLCHR =	ASA
BLKSIZE =	*n*	EOFADDR[c]	*address*
DEVICE =	READ*nn*[a] PRINTER	IOAREA2 =	*address*
DEVADDR =	SYSRDR SYSLST SYSIPT SYSOPT SYS*nnn*[a]	IOREG =	(R)
		WORKA =	YES
IOAREA1 =	*address*		

[a]The number assigned to a particular device is determined at each installation.

[b](R) specifies the register number (usually 2–11) where the information is contained.

[c]For card reader only.

WLRERR applies only to tape input files and causes a branch to the symbolic address given when a wrong-length record is read. A wrong-length record is one whose actual length is inconsistent with the length defined in BLKSIZE.

EOFADDR (end-of-file address) is the address to which a branch is taken when the end of a file has been reached. Such a condition is signaled in a card file by a card punched /* in the first two columns.

The operands described above are not the only ones that may be used with DTFSR. ALTTAPE is used to specify the symbolic name for an alternate tape unit when more than one is used for a file. CHECKPT is used whenever the CHKPT macroinstruction is used to produce *checkpoint* records. These are records produced at intervals to provide a nearby restart point, should a malfunction or other unanticipated interruption occur. An operand called CKPTREC is used whenever a tape input file contains checkpoint records among the data records. In addition, other operands are used for specialized functions in connection with specific input/output devices.

The DTFSR operands, together with their options are summarized for a card reader, punch and line printer in Table 11.1 and for magnetic tape in Table 11.2

TABLE 11.2 DTFSR OPERANDS FOR MAGNETIC TAPE UNITS

Required Operand	Options	Optional Operands	Options	Optional Operands	Options
TYPEFLE =	INPUT OUTPUT CMBND	RECSIZE =	*n* (R)[b]	LABADDR =	*address*
RECFORM =	FIXUNB FIXBLK VARUNB VARBLK UNDEF	READ =	FORWARD BACK	ERRUPT =	IGNORE SKIP *address*
		REWIND =	UNLOAD NORWD		
		CONTROL =	YES	WLRERR =	*address*
		IOAREA2 =	*address*	EOFADDR =	*address*
BLKSIZE =	*n*	IOREG =	(R)[b]	ALTTAPE =	SYS*nnn*[a]
DEVICE =	TAPE	FILABL =	STD NSTD NO	CHECKPT =	*n*[a]
DEVADDR =	SYSIPT SYSOPT SYS*nnn*[a]	WORKA =	YES	CKPTREC =	YES
				TPMARK =	NO
IOAREA1 =	*address*				

[a]See footnote a, Table 11.1.
[b]See footnote b, Table 11.1.

11.3.1.2 DTFDA

DTFDA (for Define the File For Direct Access) is used for processing disk records in random order by the *direct-access method.* By this is meant the reading or writing of records wherein the order of their use bears no relation to that in which they are stored, as is necessary with card and magnetic tape files. Some of its operands are related to those of DTFSR, while others are specific for this usage.

Operands used in the same way as in DTFSR are BLKSIZE, CONTROL, DEVICE (DISK11 is the code used to specify the 2311), IOAREA1, IOAREA2, IOREG, ERROPT, WLRERR, LABADDR, RECFORM (the only options are FIXUNB and UNDEF), RECSIZE, and TYPEFLE (the only options are INPUT and OUTPUT). Operands allowed in DTFSR but not allowed in DTFDA include READ, REWIND, DEVADDR (specified instead by a control card), and FILABLE.

The direct-access method allows records to be identified by a record key or by a record identifier. A *record key* is a field defined as part of a logical record. The DTFDA entry KEYARG is used to designate the symbolic name of this field. A key field named FIELD, for example, would require an operand of KEYARG=FIELD. All keys must be the same length, and this length is designated in DTFDA by means of the KEYLEN entry. If FIELD is a 6-byte field, therefore, the operand KEYLEN = 6 must be included. In

addition, whenever input records are to be identified by key, the operand READKEY = YES must be used, and whenever output records are to be so identified, the operand WRITEKY = YES must be used.

A *record identifier* (ID) is a 5-byte field that designates a record by its location. The first four bytes specify the track location of the record, and the fifth uniquely identifies the particular record on the track. When records are specified by ID, they must be numbered in succession, and without missing numbers, on each track. Also, the operand READID = YES must be used in conjunction with input, and the operand WRITEID = YES must be used in conjunction with output.

The track (and sometimes the record) location of a particular record to be read or written is stored in a *track-reference* field by the problem programmer. For such a field named TRFLD, the operand SEEKADR = TRFLD is included in DTFDA. The track-reference field is an 8-byte field whose contents are as follows:

Byte	*Contents*
0	Disk pack number
1-2	Cell number (for IBM 2321 Data Cell Drive)
3-4	Cylinder number
5-6	Read/write head number (bytes 3-6 together specify track identification)
7	Record number on the track

SRCHM = YES is an operand that indicates a search of multiple tracks for input/output records identified by key is to be made.

TABLE 11.3 DTFDA OPERANDS FOR DISK FILES

Operand	Option	Operand	Option	Operand	Option
TYPEFLE =	INPUT OUTPUT	LABADDR =	*address*	READKEY =	YES
		IOAREA1 =	*address*	WRITEKY =	YES
DEVICE =	DISK[1]	IOREG =	(R)	READID =	YES
IOAREA1 =	*address*	ERRUPT =	IGNORE SKIP *address*	WRITEID =	YES
CONTROL =	YES			SEEKADR =	*address*
BLKSIZE =	*n*	WLRERR =	*address*	SRCHM =	YES
RECFORM =	FIXUNB UNDEF	KEYARG =	*address*	VERIFY =	YES
		KEYLEN =	*n*		
RECSIZE =	*n* (R)				

VERIFY = YES is included if the problem programmer wants records written on the disk to be checked.

A summary of these operands is given in Table 11.3.

Other operands may be used with DTFDA as well. Some of these facilitate addition of records to the file, make status information available to the problem programmer, supply the ID of a record after read and write operations, and so forth.

11.3.1.3 Other File Definition Macro Instructions

Similar macro instructions exist for other types of file definition. One of these is DTFSN (Define the File for Synchronous Transmit-Receive Use). This macro instruction is used for processing records arriving and leaving via communication lines. A second is DTFRF (Define the File for Reference). It is used with DTFSN to allow the problem programmer to reference information in the channel command block (CCB). A third is DTFPH (Define the File for Processing by Physical IOCS*). This is used only where tape files with standard labels are to be processed by the previously introduced EXCP/WAIT pair. Still another is DTFIS (Define the File for the Indexed Sequential System of Processing). This is used for a file whose records are stored sequentially but which contain an index: such records may be accessed either sequentially or directly.

For the most general System/360 operating system, a single macro instruction is used to define files for all types of input/output operations. This macro instruction is DCB (Define a Control Block). One DCB is required for each file to be processed, and its operands contain the same sort of information as required with the other macro instructions. Such information may be categorized as (1) characteristics of the file, (2) types of macro instructions to be executed, (3) buffering choices, (4) device-dependent options, (5) exit addresses, and (6) working storage used by access-method routines.

11.3.2 General Service Macro Instructions

11.3.2.1 OPEN

OPEN is a statement that is executed in preparation for input/output operations. The file specified by symbolic name or via the symbolic address of the defined control block, depending on the operating system being used, has its label(s) checked and is positioned for reading or writing.

*Input/output Control System.

11.3.2.2 CLOSE

CLOSE is a statement that is used to terminate the processing of a file. Like OPEN, its operand field specifies the file by symbolic name or via the symbolic address of the defined control block. Both OPEN and CLOSE may specify multiple files; when there is more than one, the operands are separated by commas.

11.3.2.3 Other Services

Among the other macro instructions provided in conjunction with general input/output services are the following: LBRET (Label Return) enables control to be returned to the supervisory program after processing of special labels has been completed; FEOV (Forced End-of-Volume) gives the problem programmer the ability to discontinue processing of a particular volume of a file and to continue with the next volume; CHKPT (Checkpoint) enables the problem programmer to cause special records to be written on an output device, such records containing information required for restart at a specified point in a program. Checkpoint records are ordinarily written during long processing runs so that the sensing of error conditions necessitates rerunning the program only from the previous checkpoint, not from the beginning.

In the most general System/360 operating system, a set of five macro instructions facilitates the use during multiprogramming operations of memory areas as buffers. These are (1) GETPOOL—requests allocation of an area of memory and constructs a buffer pool in the allocated area, (2) FREEPOOL—releases a memory area previously assigned as a buffer pool; (3) BUILD—constructs a buffer pool in a memory area assigned by the programmer, (4) GETBUF—obtains a buffer area from a buffer pool, and (5) FREEBUF—returns a buffer to a pool.

11.3.3 Macro Instructions for Reading and Writing Records

Once definitions of files have been made and general service facilities have been employed, the macro instructions for reading and writing records may be used. These carry out two main functions. First, they specify exactly at what point in the program the next logical record is to be read or written; second, they enable the programmer to think in terms of reading or writing a single logical record. This latter function simplifies his task by divorcing him from the processing associated with blocking and deblocking physical records. When he initiates a read operation, for example, the next logical record is made available for his use; it may be the first of a block

of logical records just entered into memory or it may be the next in a block previously entered into memory. Similarly, the programmer is relieved of the tasks associated with volume switching by these macro instructions, unless he chooses to maintain control of this operation with the FEOV macro instruction.

11.3.3.1 READ

This macro instruction makes the next logical record available for processing. The operands, whose use depends on (1) the operating system of which it is a part, (2) the manner in which the file is organized, and (3) the access method being specified, include either the name of the file or the name of the macro instruction defining the file and certain other items. Among these additional operands may be STR (specifying synchronous transmit-receive communications devices), KEY (specifying that the key field is to be used to locate the record), IS (specifying that the file is organized in the indexed sequential manner), D (specifying the direct access method), and so on.

File organization falls into five classes. A *sequential* file is one whose records are organized solely on the basis of their successive physical positions. A *partitioned* file is one that is divided into sequentially organized members, each with an addressable name. An *indexed sequential* file is one whose records are ordered by keys that precede each block of data. A *direct* file is one whose records are arranged on direct-access volumes in any manner chosen by the programmer. A *telecommunications* file is one whose records are messages sent or received on communications facilities.

File access may be *sequential* and/or *direct* (random), depending on the physical characteristics of the input/output device. Sequential access is access governed by the order in which records are encountered, as in a magnetic tape file or card file. Direct access is access independent of the other records in the file, as in a magnetic disk file. (Access to a file on magnetic disks may also be sequential, however.)

11.3.3.2 WRITE

This macro instruction places the logical record specified in an output file. Its operands are similar to those of READ. The execution of WRITE, where blocked records are being used, may not cause an actual output operation but only the placement of the logical record in a buffer which will not be written out until full. When an output operation does occur—and this is also true of both READ and the input/output macro instructions below—a channel program is executed. This channel program in skeletal form is part of the macro instruction and is filled out according to the items specified in the operands. In addition, an EXCP macro instruction is em-

bedded within the *higher level* macro instruction and is used to initiate the operation just as if it had been inserted directly by the programmer.

11.3.3.3 GET and PUT

These macro instructions are very similar to READ and WRITE, respectively, but the differences between READ and GET and between WRITE and PUT are a function of the operating systems in which they appear. In one, READ and WRITE are used for processing telecommunications files, and GET and PUT are used for sequential files; in a second, READ and WRITE are used for direct access, and GET and PUT are used for sequential access; in a third, READ and WRITE are used for processing all files, and GET and PUT are used for processing sequential, indexed sequential, and telecommunications files as well, etc.

In the last case specified in the preceding paragraph, both the READ/WRITE pair and the GET/PUT pair of macro instructions may be used for processing files organized as sequential, indexed sequential, and telecommunications files. In this instance, which occurs in the most general System/360 operating system, the two sets are not equivalent. READ/WRITE is used for *basic access*, and GET/PUT is used for *queued access*. Both the basic and queued access macro instructions carry out input/output operations; the latter, in addition, provide anticipatory buffering and synchronization of input/output with CPU processing.

System-controlled anticipatory buffering permits the programmer to use as many input or output buffers as he needs without issuing multiple GET or PUT macro instructions to fill or to empty buffers. The system automatically sets up the requested number of buffers and provides overlap of input/output operations with CPU processing by automatically filling or emptying buffers at the earliest possible times. Thus more than one input or output block normally is in memory at the same time to prevent input/output operations from delaying record processing.

Automatic synchronization of input/output operations with processing operations relieves the programmer from testing for completion of input/output operations, input/output errors, or exceptional conditions. After a GET or PUT macro instruction is issued, the system does not return control to the problem program until an input area is filled or an output area is available. Exits to error correction and end-of-volume or end-of-file procedures are automatically taken when the corresponding condition occurs.

EXAMPLE 49

To illustrate the basic use of the GET/PUT pair, we shall set up DTFSR's for a card reader (let's say a 2540) and for a printer. The input area will be called TO and will be sized to hold an 80-character record (i.e., the contents of a card). Similarly, the output area, called FROM, will be sized for a 120-character record plus an additional byte, placed ahead of each record, for carriage control information. We

shall use the names CRDNICK and PRTNICK for the input and output files respectively.

Solution Program

```
Name      Operation   Operand                    Comment
          START       5000
CRDNICK   DTFSR       TYPEFLE=INPUT,             This defines our input
                      RECFORM=FIXUNB,              file.
                      BLKSIZE=80,
                      DEVICE=READ40,
                      DEVADDR=SYSIPT,
                      IOAREA1=TO,
                      EOFADDR=NOMORE
PRTNICK   DTFSR       TYPEFLE=OUTPUT,            This defines our output
                      RECFORM=FIXUNB,              file.
                      DEVADDR=SYSLST,
                      IOAREA1=FROM,
                      CTLCHR=ASA
          .
PROG      BALR        15,0
          USING       *,15
          OPEN        CRDNICK,PRTNICK            Files are available for
                                                   use.
          GET         CRDNICK                    This will read the next
                                                   card into TO.
          .
          instructions                           This refers to whatever
                                                   processing the pro-
                                                   gram does including
                                                   the construction of
                                                   the output record,
                                                   carriage control char-
                                                   acter and all in
                                                   FROM.
          PUT         PRTNICK
```

```
Name      Operation   Operand            Comment
NOMORE    CLOSE       CRDNICK.PRTNICK    This is the instruction
                                           referred to in
                                           EOFADDR of
                                           CRDNICK.
          EOJ
TO        DS          CL80
FROM      DS          CL121
          END         PROG
```

11.3.3.4 PUTX

This macro instruction is used with certain file organizations to cause a logical record contained in an input file buffer to be written as the next sequential record of an output file. In certain cases, this is carried out with no movement of data in memory.

11.3.3.5 RELSE

RELSE (Release) is used in conjunction with blocked input records. It allows the programmer to skip the remaining logical records in a block and to continue processing with the first record of the next block.

11.3.3.6 TRUNC

TRUNC (for Truncate) is used in conjunction with blocked output records. It allows the programmer to write a short block of records so that the next block may start with a record in a new category.

11.3.3.7 CNTRL

CNTRL (Control) is used to cause actions specific to special types of devices. Among these are card-reader stacker selection, line space on printer, carriage-control tape skip on printer, forward or backward spacing of magnetic tape over a specified number of blocks, etc.

11.3.3.8 PRTOV

PRTOV (Printer Carriage Overflow) is used to control the page format on a printer by giving the programmer the ability to test the carriage-control tape to determine if an overflow condition exists.

As with previous categories of macro instructions, it has been possible to include here only a sampling of the language features available to the

programmer and within this subset to include only an outline of the way they are used. The scope of this book can include only these fundamentals because of the large number of combinations of System/360 operating systems, file organization and access methods, input/output macro instructions, and special items related to specific characteristics of individual input/output devices.

FURTHER READING

1. *Basic Operating System—Macro Definition Language*, Form C24-3364. IBM Corp., White Plains, N.Y.
2. *Tape/Disk Operating Systems—Extended Utility Macros Specifications*, Form C24-5042. IBM Corp., White Plains, N.Y.
3. *Tape/Disk Operating Systems—Extended Supervisor and Input/Output Macros*, Form C24-5035, C24-5307. IBM Corp., White Plains, N.Y.
4. *Operating System—Supervisor and Data Management Macroinstructions*, Form C28-6647. IBM Corp., White Plains, N.Y.

Chapter 12

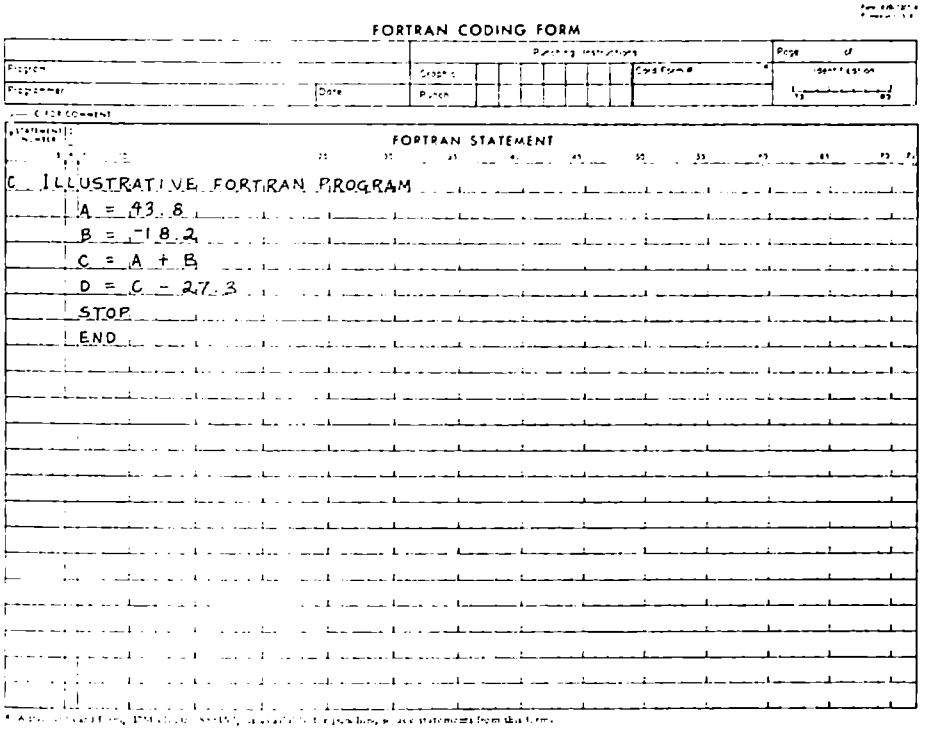
FORTRAN Programming

FORTRAN (derived from FORmula TRANslation), like Assembler Language, consists of statements which the problem programmer uses as required to express his problems. These statements, punched into cards in a format adjusted to the nature of the language, bear no relation to the machine language instructions of System/360. Instead, they are designed to resemble the notation used for conventional algebra and arithmetic. The version of FORTRAN used for System/360, that is, the language features and the accompanying compiler, represents a modification and extension of earlier forms and is known as FORTRAN IV. A wide array of compilers exist for FORTRAN IV, making it (or subsets of it) available for more types of machines than any other programming language.

12.1 BASIC CODING CONVENTIONS

FORTRAN source programs are ordinarily written on a standard FORTRAN Coding Form (Figure 12.1) from which the source program is punched, one card per line. Many of the ground rules associated with writing FORTRAN programs can be explained in reference to this figure.

First, note that only columns 1-72 of each card are used for writing FORTRAN statements. Columns 73-80 are ignored by the FORTRAN compiler and may or may not be used by the problem programmer. These columns


FORTRAN CODING FORM

FORTRAN STATEMENT

C ILLUSTRATIVE FORTRAN PROGRAM
A = 43.8
B = -18.2
C = A + B
D = C - 27.3
STOP
END

Figure 12.1 FORTRAN coding form

are commonly used for program identification and/or card numbering.

FORTRAN statements proper may begin anywhere but not before column 7; columns 1-6 are ordinarily blank. There are three instances in which one or more columns in the range 1-6 are not blank: comment cards, continuation cards, and statement numbers.

A *comment card* has the letter C in column 1 and may have any allowable characters (all alphabetic characters, all numeric digits, and the special characters $ - + / = .) * , (' & blank (b) in columns 2-72. These cards are listed with the source program but are not otherwise processed by the FORTRAN compiler. As with Assembler Language comment cards, they are for the convenience of the programmer and those who may wish to read his source program.

A *continuation card* has any character, except a blank or a zero, in column 6. It provides the capability, in case a statement is too long to fit in columns 7-72, of using more than one line per statement. The first line of a statement always has a blank or zero in column 6; each continuation card, up to 19, has some other character in that column.

For branching and other reference purposes, each FORTRAN statement may be assigned a *statement number* by the programmer. Such a number, which must be an unsigned decimal integer, is from one through five digits in length and is entered in columns 1-5. Should a statement number be

less than five digits in length, its placement in columns 1-5 is arbitrary, with the exception that the digits must be contiguous.

The statements, A = 43.8 and B = −18.2 (see Figure 12.1) are analogous to DC statements in Assembler Language. In each case, a constant is defined in terms of a symbolic address. The types of definable constants in FORTRAN are discussed below. In this instance A and B are short-operand, floating-point constants. The statement C = A + B specifies that the contents of locations A and B are to be added and that their sum, in short-operand, floating-point form, is to be stored in location C. This statement also serves the same purpose as an Assembler Language DS in defining C. The following statement D = C − 27.3 is included to show that a constant may be incorporated into a statement without being defined, as were A and B. STOP is equivalent to the Assembler Language EOJ, which sends control back to the Supervisor at the end of object program execution. END is equivalent to the Assembler Language END, which notifies the source language processor that there are no further source language statements.

There are a few additional facets of the illustrative sequence in Figure 12.1 that shed light on the nature of FORTRAN programming. First, like those of Assembler Language, the statements are normally executed in sequence. Special statements, none of which are shown in this example, are used to cause breaks in this sequence. Second, FORTRAN statements are designed to be similar to those in English and mathematics. This similarity is not an identity, however, and certain differences must be noted. Writing STOP as S T O P would cause no misunderstanding to the driver approaching an intersection; however, blanks, commas, or other characters may not be inserted within the symbols, constants, or keywords used in FORTRAN.

FORTRAN arithmetic statements such as D = C − 27.3 and C = A + B have the appearance of mathematical equations and ordinarily they may be treated as such. They are not true equations, however, and should be considered in the following way: (1) the numerical value of the expression on the right side of the equals sign is computed, and (2) this value is then inserted as the new value of the variable on the left side of the equals sign. Thus FORTRAN statements such as X = X + 1 and N = N − 4, which are inconsistent in mathematics, are quite meaningful in FORTRAN.

Lastly, every symbol, constant, or keyword used in FORTRAN statements must be followed by one or more of the following: + − * / =) (. , or a blank or by column 73. One or more blanks, however, may be added to any of these special characters to improve readability without interfering with the functioning of the FORTRAN compiler.

12.2 NUMERICAL VARIABLES AND ARITHMETIC EXPRESSIONS

The FORTRAN programmer can exercise great flexibility in specifying and manipulating numeric data. By assigning meaningful names of his own choosing and combining constants and variables into arbitrarily

complex expressions, it is possible to specify extensive operations in single statements.

12.2.1 Naming FORTRAN Variables

The assignment of variable names is governed by a small number of relatively simple rules:

1. Variable names may be from one through six characters in length.
2. The first character of a variable name must be alphabetic.
3. The characters that may be used for variable names are the alphabetic characters A–Z and the numeric digits 0–9. In addition, the special character $ is considered to be an alphabetic character in this context and may be used in any way that the letters A–Z are used.

All combinations of characters formed according to rules 1, 2, and 3 are valid except a group reserved by the FORTRAN compiler for special purposes. These are listed in the table below and are discussed in the context of programmed examples in this and subsequent chapters.

ABS	DO	FORMAT	PAUSE
BACKSPACE	DOUBLE	FUNCTION	READ
CALL	DSIGN	GO	REAL
COMMON	END	GOTO	RETURN
CONTINUE	ENDFILE	IABS	REWIND
DABS	EQUIVALENCE	IDIM	SIGN
DBLE	EXIT	IF	SNGL
DEFINE	EXTERNAL	IFIX	STOP
DIM	FIND	INTEGER	SUBROUTINE
DIMENSION	FLOAT	ISIGN	WRITE
DFLOAT			

12.2.2 Properties of Numeric Variables

We shall be dealing with three basic types of numeric variables: *integer* variables, *real** variables, and *double precision* variables. They correspond, respectively, to the 32-bit fixed-point words, 32-bit floating-point words, and 64-bit floating-point doublewords previously encountered in machine language and Assembler Language.

Integer and real variables may be specified implicitly; all three types may be specified explicitly. Implicit specification is effected by means of the initial (it may be the only) character of a variable's name. If the first character is I, J, K, L, M, or N, the variable is an integer; otherwise, it is real. The choice of these letters is not fortuitous; they are conventionally used in

*The use of the term *real* should not be associated with its meaning in mathematics, since both real variables and double precision variables are mathematically equivalent according to a mathematical classification. In fact, their classification as *rationals* would be more precise; digital computers, being finite, cannot represent irrational numbers numerically because an infinite number of digits would be required.

mathematics as subscripts, and subscripts are integers in FORTRAN as in mathematics.

Variables are explicitly specified by means of special FORTRAN statements provided for this purpose. In the following statements

Statement Number	*FORTRAN Statement*
	INTEGER A, I, RESULT
	REAL X, Y, MEAN
	DOUBLE PRECISION SUM, NUMBER

A, I, and RESULT are specified as integer variables, X, Y, and MEAN as real variables, and SUM and NUMBER as double precision variables. When explicit specification statements are used, they must be placed ahead of the first executable statement of the program.

12.2.3 Constants

Numeric constants are classified as *integer*, *real* or *double precision*, in parallel with the three classes of variables. *Integer constants* are positive, negative, or zero values whose maximum magnitude is 2147483647 ($2^{31} - 1$) and which do not include a decimal point. The following are examples of integer constants:

```
2
+2
-345
0
1000000
-1234567
+70532
```

Real constants can be positive, negative, or zero. They may consist of one through seven decimal digits which include a decimal point. A real constant may optionally be followed by the letter E and one or two positive or negative digits to denote multiplication by a power of 10. The following are examples of real constants:

2.0	
-2.	
+2.3	
8.3E6	(8.3×10^{6})
-43.36E-12	(-43.36×10^{-12})
0.	
9.38E-08	(9.38×10^{-8})

Double-precision constants may be positive, negative, or zero and must include a decimal point. They may be of one through seven decimal digits followed by the letter D and one or two positive or negative digits as with

the E notation above, or may be of eight through sixteen decimal digits optionally followed by the D notation. The approximate range of such constants is from $\pm 10^{-75}$ to $\pm 10^{+75}$, and 0. The following are examples of double-precision constants:

1.7D6	(1.7×10^{6})
−1234.0000	
0.0D0	(0.)
483.60103479D−18	$(483.60103479 \times 10^{-18})$

12.2.4 Arithmetic Expressions

Arithmetic expressions in FORTRAN closely resemble their algebraic counterparts. That is, they are constructed from combinations of variables and/or constants connected by arithmetic operators. The basic difference is that all operations must be explicitly indicated in FORTRAN expressions: they cannot be implied.

Five basic operators are provided:

Operation	*Symbol*
addition	+
subtraction	−
multiplication	*
division	/
exponentiation	**

If A and B are two variables, the combination 2AB or 2(A)(B) is clear in algebraic notation but is illegal in FORTRAN. Instead, we must say 2.*A*B.

The FORTRAN compiler processes arithmetic expressions by working on them from left to right in several stages, as dictated by the presence or absence of certain operators. All exponentiations are performed first, followed by all multiplications and divisions. Additions and subtractions are done last.

If the programmer has any doubts about the order in which the components of a given expression are handled, he may remove those doubts by using parentheses. This will force FORTRAN to evaluate the parenthesized components first. Some examples are shown below:

FORTRAN Expression	*Algebraic Equivalent*
X + 4.*Y/Z	$X + \frac{4Y}{Z}$
(X + 4.*Y)/Z	$\frac{X + 4Y}{Z}$
X**2 + 6.*Y/3.2*Z	$X^2 + \frac{6Y}{3.2}(Z)$

FORTRAN Expression	*Algebraic Equivalent*
X**2 + 6.*Y/(3.2*Z)	$X^2 + \frac{6Y}{3.2Z}$
(X**2 + 6.*Y)/(3.2*Z) + 5.0	$\frac{X^2 + 6Y}{3.2Z} + 5$

EXAMPLE 50

Given A = 4.8, B = 7.7, and C = −2.3. Compute DISC as $B^2 - 4AC$.

Solution Program

Statement Number	*FORTRAN Statement*
	A = 4.8
	B = 7.7
	C = −2.3
	DISC = B**2 − 4.*A*C
	STOP
	END

Computation can be programmed using all three types of variables and constants, but real and double-precision numbers are ordinarily used to gain the advantages of floating-point arithmetic. Integers are most often used as exponents and subscripts although, as was stated, there is no restriction against wider use. Real and double-precision numbers may be used as exponents, but where the exponents are integral, there may be a loss in efficiency. In the Solution Program for Example 50, for instance, B^2 was programmed as B**2, but a real exponent (such as B**2.0) could have been used. Both give correct results, though the translations of the FORTRAN compiler are widely different. The first method causes B^2 to be calculated as the product of B times itself. The second method, which recognizes that real exponents do not have to be integral, causes B^2 to be calculated as the antilogarithm of the product of the exponent and the logarithm of B. There is a substantial difference in the amount of computation involved.

Division of two integers is also to be avoided because the result is an integer with no rounding. Thus, 12/7 produces 1 and 4/5 produces 0. Similarly, truncation occurs when a real value is assigned to an integer variable. Thus the sequence

```
A = 17.8
I = A
```

will produce a value of 17 in I. By the same token,

```
I = 20
A = I/3
```

will produce a 6.0 in A, because truncation precedes conversion from integer to real form.

EXAMPLE 51

Given N1 = 429, N2 = 250, N3 = 350, and N4 = 275, compute the sum as an integer; convert this sum to a real variable; and compute the mean of the four data.

Solution Program

Statement Number *FORTRAN Statement*

```
      REAL MEAN
      N1 = 429
      N2 = 250
      N3 = 350
      N4 = 275
      NSUM = N1 + N2 + N3 + N4
      SUM = NSUM
      MEAN = SUM/4.
      STOP
      END
```

After the specification of MEAN as a real variable, the next five statements compute the sum of the data as an integer. An alternate means of doing this is to use the single statement, NSUM = 429 + 250 + 350 + 275. The next statement, SUM = NSUM, converts NSUM to real number form and sets SUM to this value. This conversion could be combined with the preceding five statements as SUM = 429 + 250 + 350 + 275 if the computation of the sum as an integer were not specified in this example. Finally, MEAN is set to one-fourth the sum to complete the calculation.

12.2.5 Built-in Functions

Commonly used functions and operations, such as the square root and logarithm, are specially preprogrammed and included as part of the FORTRAN compiler. Each of these is given a reference name associated with its usage. These names are part of the language, just like other keywords, and may be used as if they were single instructions.

A function is used by specifying its name, followed by the information to be worked on (*argument(s)*) enclosed in parentheses. For example, EXP is the keyword for a built-in function that calculates powers of *e*, the base of natural logarithms. Thus, assuming X has a numerical value, the statement

```
Y = 2.*EXP(X)
```

will cause the value $2e^{x}$ to be placed in Y.

The argument need not be restricted to a simple variable. Any legitimate expression may be used, regardless of its complexity, as long as it works down to a single value. For instance, it is perfectly legal to write statements

such as

```
Y = 2.*EXP(3.*X - 6.6*X**2)
```

or even

```
Y = 2.*EXP(3.*X - 6.6*EXP(.2*X**2))
```

Some additional built-in functions are listed in Table 12.1

EXAMPLE 52

Compute $Y = \sqrt{X^2 + A^2}$, where X = 0.000000083 and A = 0.00000000378.

Solution Program

Statement Number	*FORTRAN Statement*
	X = 8.3E−8
	A = 3.78E−9
	Y = (X**2 + A**2)**.5
	STOP
	END

This example illustrates the use of the E notation to express real numbers that would otherwise be somewhat unwieldy. It also illustrates the use of a fractional exponent. While the use of the fraction, .5, to specify the extraction of square root will produce the correct result, there is an alternate way. The third statement of the Solution Program could be written as

```
Y = SQRT (X**2 + A**2)
```

If we wanted to work with X, Y, and A as double-precision values, our Solution Program would look as follows:

Statement Number	*FORTRAN Statement*
	DOUBLE PRECISION X, A, Y
	X = 8.3D−8
	A = 3.78D−9
	Y = DSQRT (X**2 + A**2)
	STOP
	END

Note that the double-precision variables must be defined as such; the values of such variables are expressed using the D notation, and the DSQRT function (Table 12.1), which is preprogrammed to extract the double-precision square root of double-precision arguments, is used instead of SQRT.

Numerous variations of this program can be devised, all of which will produce the same correct result. The reason for this apparent vagueness is that arithmetic expressions may include mixtures of integer, real, and double-precision variables and/or constants; yet, the rule for combining such variables and/or constants often masks combinations that were originally different. The rule is this: An arithmetic expression, which is the right-hand side of an arithmetic assignment statement,

TABLE 12.1 SOME FORTRAN BUILT-IN FUNCTIONS. X1, X2, I1, I2, D1, D2 REPRESENT PAIRS OF REAL, INTEGER, AND DOUBLE-PRECISION ARGUMENTS, RESPECTIVELY

Name	Result	Usage	Name	Result	Usage
EXP	e^{X1}	EXP (X1)	IABS	|I1|	IABS (I1)
DEXP	e^{D1}	DEXP (D1)	ABS	|X1|	ABS (X1)
SQRT	$\sqrt{X_1}$	SQRT (X1)	DABS	|D1|	DABS (D1)
DSQRT	$\sqrt{D_1}$	DSQRT (D1)	INT	largest integer ≤ |X1|	INT (I1)
ALOGIO	$\log_{10}X1$	ALOGIO (X1)	AINT	largest integer ≤ |X1|	AINT (X1)
DLOGIO	$\log_{10}D1$	DLOGIO (D1)	IDINT	largest integer ≤ |D1|	IDIM (D1)
ALOG	ln X1	ALOG (X1)	AMAX0	max (I1,I2, etc.) (result is REAL)	AMAX0 (I1,I2, etc.)
DLOG	ln D1	DLOG (D1)			
SIN[a]	sin X1	SIN (X1)	AMAX1	max (X1,X2, etc.) (result is REAL)	AMAX1 (X1,X2, etc.)
DSIN[a]	sin D1	DSIN (D1)			
COS[a]	cos X1	COS (X1)	MAX0	max (I1,I2, etc.) (result is INTEGER)	MAX (I1,I2, etc.)
DCOS[a]	cos D1	DCOS (D1)			
ATAN	$\tan^{-1}$ X1	ATAN (X1)	MAX1	max (X1,X2, etc.) (result is REAL)	MAX (X1,X2, etc.)
ATAN2	$\tan^{-1}$ D1	ATAN2 (D1)			
TANH	tanh X1	TANH (X1)	DMAX1	max (D1,D2, etc.) (result is DOUBLE)	DMAX1 (D1,D2, etc.)
DTANH	tanh D1	DTANH (D1)			
MOD[b]	Remainder of $\frac{I1}{I2}$	MOD (I1,I2)	AMIN0, AMIN1, MIN0, MIN1, DMIN1	Analogous to corresponding MAX functions	
AMOD[b]	Remainder of $\frac{X1}{X2}$	AMOD (X1,X2)			
DMOD[b]	Remainder of $\frac{D1}{D2}$	DMOD (D1,D2)			
DIM[c]	X1 − min (X1,X2)	DIM (X1,X2)	FLOAT	I1 to X1	FLOAT (X1)
IDIM[c]	I1 − min (I1,I2)	IDIM (I1,I2)	DFLOAT	I1 to D1	DFLOAT (I1)
SIGN[c]	sign (X2) * X1	SIGN (X1,X2)	IFIX	X1 to I1	IFIX (X1)
ISIGN[c]	sign (I2) * I1	ISIGN (I1,I2)	HFIX	D1 to I1	HFIX (D1)
DSIGN[c]	sign (D2) * D1	DSIGN (D1,D2)	DBLE	X1 to D1	DBLE (X1)

[a] Argument is assumed to be in radians
[b] Remainder is expressed as an integer; thus, MOD (13-5) = 3, not 3/5
[c] Not available in basic versions

produces a double-precision result if any of its operands is in double-precision mode, a real result if none of the operands is double-precision and at least one is real, and an integer result if all operands are integers. Thus the program immediately above could produce the same result if written as follows:

Statement Number	*FORTRAN Statement*
	DOUBLE PRECISION Y
	X = 8.3E−8
	A = 3.78E−9
	Y = DSQRT (X**2 + A**2)
	STOP
	END

12.2.6 Arrays

A collection of values (usually called a *matrix* or an *array*) may be referred to by a single name as in mathematics. Each member of such a collection (an *element*) is distinguished from all others by a set of subscripts which locate the element uniquely.

12.2.6.1 Dimensionality

The *dimensionality* of an array refers to the number of criteria that must be specified to identify any element uniquely. More directly, the dimensionality is reflected in the number of subscripts that must be appended to an element of a given array.

The simplest type of array (one-dimensional) is merely a list of elements, each of which requires a single subscript. For example, the six-element array X is organized as shown:

X(1)	X(2)	X(3)	X(4)	X(5)	X(6)

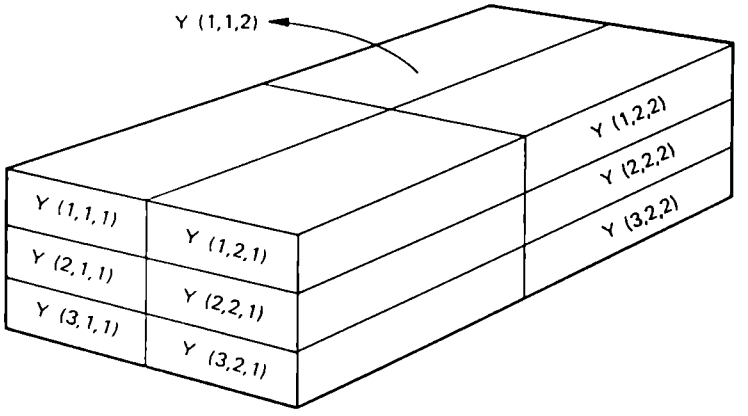
A two-dimensional array is organized into a rectangular network consisting of horizontal rows and vertical columns. Thus a six-element array X, when organized into three rows and two columns, would appear as

X(1,1)	X(1,2)
X(2,1)	X(2,2)
X(3,1)	X(3,2)

As shown here, a single element is specified by two subscripts, with the row subscript given first. When a two-dimensional FORTRAN array is stored internally, its elements are arranged by columns. Thus for the example

shown above, the elements X(1,1), X(2,1), X(3,1), X(1,2), X(2,2), and X(3,2) would be stored in consecutive locations.

A three-dimensional array (the highest allowable dimensionality in most versions of FORTRAN) is organized into rows, columns, and blocks. Thus a twelve-element array Y consisting of three rows, two columns and two blocks would be organized as follows:



These elements would be stored in the following order: Y(1,1,1), Y(1,1,2), Y(1,2,1), Y(1,2,2), Y(2,1,1), Y(2,1,2), Y(2,2,1), Y(2,2,2), Y(3,1,1), Y(3,1,2), Y(3,2,1), Y(3,2,2).*

12.2.6.2 Subscripts

Subscripts are always integers. They may be variables, constants, or certain combinations of these. The seven forms that a subscript may have are shown in the following list, where v represents an unsigned, nonsubscripted, integer variable, and c and k represent unsigned integer constants.

Subscript Form	*Example of Use of this Form*
v	A(I)
c	A(2)
$v + c$	A(I + 2)
$v - c$	A(I − 2)
$c * v$	A(2 * I)
$c * v + k$	A(2 * I + 7)
$c * v - k$	A(2 * I − 7)

There is a difference in range between subscripts and other integers. Subscripts must always be greater than zero and less than 32767; other integer constants and variables may range from −2147483647 to +2147483647.

*See Sec. 14.7, Par. 3.

12.2.6.3
The DIMENSION Statement

Storage for arrays is allocated by the DIMENSION statement. For example,

```
DIMENSION A(3,5), B(3)
```

specifies that A has fifteen elements, arranged in three rows and five columns, and that B has three elements. The type of element is specified in the same manner as are nonsubscripted variables: that is, implicitly via the first character of its symbolic name or explicitly via an INTEGER, a REAL, or a DOUBLE PRECISION specification. In this example, the statement REAL A(3,5), B(3) could have been used instead of the DIMENSION statement.

EXAMPLE 53

A rectangular matrix with three rows and five columns is stored in memory. It is called A and its elements are real variables. Also, a vector (linear array) called B, containing three real elements, is stored in memory. Perform the following calculations, which are specified using conventional subscript notation:

(1) add B_1 to $A_{1,2}$;
(2) subtract B_2 from $A_{2,4}$; and divide $A_{4,5}$ by B_3

Solution Program

Statement Number	*FORTRAN Statement*
	DIMENSION A(3,5), B(3)
	A(1,2) = A(1,2) + B(1)
	A(2,4) = A(2,4) − B(2)
	A(4,5) = A(4,5)/B(3)
	STOP
	END

12.3
BASIC SEQUENCE CHANGING

FORTRAN provides the programmer with access to a wide variety of comparison and branching capabilities. By using relatively simple statements, it is possible to specify very complex decision structures.

12.3.1
The Unconditional GO TO Statement

A simple branch, corresponding directly to the Assembler Language statement

```
BC      15,operand
```

is effected by the unconditional GO TO statement, whose general form is

GO TO *n*

where *n* is the statement number attached to the desired destination.

12.3.2 The Computed GO TO Statement

The basic GO TO statement may be expanded to provide a choice of destinations based on some internal value. To do this, we write

GO TO (*n*1, *n*2, *n*3. . . .), *v*

where *v* is an integer variable whose value dictates the statement number selected for the destination. The value of *v* is used to determine the *sequence number* of the statement in the list. Thus if we wrote

GO TO (6, 4, 6, 2, 10, 7, 33, 8), N

the relationship between N and the GO TO's destination would be as follows:

N	GO TO *would act as if it read*
1	GO TO 6
2	GO TO 4
3	GO TO 6
5	GO TO 10
7	GO TO 33

If the indicating variable's value happens to be below 1 or greater than the number of alternatives (above 8 in this case) the entire statement will be ignored.

12.3.3 The Arithmetic IF Statement

The basic mechanism for comparison and conditional branching is embodied in the arithmetic IF statement. This provides the capability to evaluate an arithmetic expression and to branch to one of three statements depending on whether the expression's value is negative, zero, or positive. Its general form is

IF (*expression*) *n*1, *n*2, *n*3

For example, the statement

IF (X − 4) 12, 14, 8

says, in essence: Branch to statement 12 if (X − 4) is less than zero (i.e., if $X < 4$); branch to statement 14 if (X − 4) is zero (i.e., if $X = 4$); branch to statement 8 if (X − 4) exceeds zero (that is, if $X > 4$).

12.3.4
The STOP *n* and PAUSE *n* Statements

FORTRAN offers a number of statement types which serve as debugging aids. One of these is the STOP *n* statement where *n* is an integer constant from 1-5 digits long. This operates like the ordinary STOP statement except that the *n* value is printed before activities cease. The programmer can place several of these statements, with different *n*'s, at various strategic places in his program, so he can analyze the occurrence of unexpected stops (before the logical end of the program).

EXAMPLE 54

Solve the general quadratic equation, $AX^2 + BX + C = 0$, given that the real coefficients, A, B, and C, are in memory.

Solution Procedure

1. Test the sign of the discriminant, $B^2 - 4AC$.
2. If positive, the real and unequal roots are $(-B \pm \sqrt{B^2 - 4AC})/2A$.
3. If zero, the real and equal roots are $-B/2A$.
4. If negative, the complex and unequal roots are $(-B \pm i\sqrt{4AC - B^2})/2A$ where i is the imaginary unit.

Solution Program

Statement Number	*FORTRAN Statement*
C	R1 AND I1 ARE THE REAL AND IMAGINARY PARTS OF THE FIRST ROOT
C	R2 AND I2 ARE THE REAL AND IMAGINARY PARTS OF THE SECOND ROOT
C	
	REAL R1, I1, R2, I2
	IF (B**2 − 4.*A*C) 4, 1, 2
2	I1 = 0
	I2 = 0
	R1 = (−B + SQRT (B**2 = 4.*A*C))/(2.*A)
	R2 = (−B − SQRT (B**2 − 4.*A*C))/(2.*A)
	GO TO 10
4	R1 = (−B/(2.*A)
	R2 = R1
	I1 = (SQRT(4.*A*C − B**2))/(2.*A)
	I2 = −I1
	GO TO 10
1	R1 = (−B)/(2.*A)
	R2 = R1
	I1 = 0
	I2 = 0
10	STOP 12345
	END

An alternate solution can be programmed in fewer statements:

Alternate Solution Program

```
Statement Number   FORTRAN Statement
C                  R1 AND I1 ARE THE REAL AND IMAGINARY
                   PARTS OF THE FIRST ROOT
C                  R2 AND I2 ARE THE REAL AND IMAGINARY
                   PARTS OF THE SECOND ROOT
C
                   REAL R1, I1, R2, I2
                   R1 = (-B)/(2.*A)
                   R2 = R1
                   I1 = 0
                   I2 = 0
                   IF (B**2 - 4.*A*C) 4,1,2
4                  I1 = (SQRT (4.*A*C - B**2) )/(2.*A)
                   I2 = I1
                   GO TO 1
2                  R1 = R1 + (SQRT (B**2 - 4.*A*C) )/(2.*A)
                   R2 = R2 = (SQRT (B**2 - 4.*A*C) )/(2.*A)
1                  STOP 12345
                   END
```

The same thing can be done with the PAUSE *n* statement, except that this specifically causes the program to stop at a designated point. By using various values, the programmer can establish the fact that execution of a given part of his program has occurred. Operation will resume after a PAUSE *n* statement when the machine's START button is pushed. Once debugging is complete, the STOPS and PAUSES can be removed and the program can be recompiled without them.

EXAMPLE 55

A real variable, x, is in memory. If its absolute value is less than 1, compute 10 terms of

$$\tan^{-1}x = x - 1/3x^3 + 1/5x^5 - 1/7x^7 + \cdots$$

and if its absolute value is 1 or greater, notify the operator.

Solution Procedure

After testing the absolute value of x, form a loop which adds one term to the accumulating sum upon each iteration.

Solution Program

```
Statement Number   FORTRAN Statement
                   IF (X**2 - 1.) 8, 10, 10
10                 PAUSE 11111
```

```
Statement Number   FORTRAN Statement
                   GO TO 20
8                  ARCTAN = X
                   TOP = X
                   BOTTOM = 1.
                   Y = 1.
                   DO 9 I = 1, 9
                   TOP = TOP * X**2
                   BOTTOM = BOTTOM + 2.
                   Y = -Y
9                  ARCTAN = ARCTAN + (Y * TOP)/BOTTOM
20                 STOP
                   END
```

The procedure used in this example is applicable to all series computations. The criterion for termination, however, is not often a fixed number of terms. More often a computation is continued until some meaningful criterion, such as a given number of significant digits, is met. The DO statement is explained below.

12.4 THE DO STATEMENT

One of FORTRAN's most powerful features greatly facilitates the construction of loops. By using the DO statement it is possible to let the compiler automatically set up the instructions for handling most of the bookkeeping. To see how this works, we shall take a look at a manually constructed loop.

EXAMPLE 56

Two vectors (one-dimensional arrays) of double precision numbers are in memory. Each has 100 elements. Compute the scalar product (dot product) of these vectors.

Solution Procedure

If A and B are the two vectors, then the scalar product is defined as the sum of the 100 products formed by multiplying together the corresponding elements of A and B. That is,

$$SP = \sum_{i=1}^{100} A_i B_i = A_1B_1 + A_2B_2 + \cdots + A_{100}B_{100}$$

1. Set SP and a subscript variable, say N, to zero.
2. Form a loop which (a) increases N by 1, (b) augments SP by A(N) * B(N), and (c) tests whether N has reached 100.
3. When N is less than 100, the loop is repeated; when N is equal to 100, the current value of SP is the scalar product.

Manual Solution Program

Statement Number	*FORTRAN Statement*
	DOUBLE PRECISION A (100), B(100), SP
	SP = 0.
	N = 0
1	N = N + 1
	SP = SP + A(N) * B(N)
	IF (N − 100) 1, 8, 8
8	STOP
	END

12.4.1 Basic Forms

Automatic looping mechanisms can be set up by the DO statement, whose basic form is

DO *n* *v*1 = *n*1, *n*2

n is the number of a statement located at some subsequent point in the program. The DO statement initiates a sequence of instructions which constitute a DO *loop*. The statement numbered *n* represents the end of a DO loop and therefore *n* specifies the loop's *range*. *v*1 is an integer variable (called an *index variable*) which FORTRAN will use to keep track of the number of times the statements in the loop are repeated. *n*1, which can be an integer constant or variable, specifies the *starting value* for *v*1. Similarly, *n*2 defines the *limiting value* for *v*1. Thus, when the DO statement is encountered, *v*1 is set to *n*1 and all of the statements through statement *n* are executed. *v*1 is increased by 1 and tested against *n*2. If it does not exceed *n*2, the statements in the loop are executed again. The process is repeated until *v*1 exceeds *n*2 by 1, at which point the program automatically branches to the statement following the end of the loop.

To examine this action for a particular example, we shall reprogram Example 56 using a DO loop.

Second Solution Program

Statement Number	*FORTRAN Statement*
	DOUBLE PRECISION A(100), B(100), SP
	SP = 0.
	DO 1 N = 1, 100
1	SP = SP + A(N) * B(N)
	STOP
	END

12.4.2 The CONTINUE Statement

Not every statement within the range of a DO statement need be executed on every iteration or, indeed, on any iteration. For correct indexing of the loop to take place, however, the last statement in the range must be executed on every iteration. Sometimes there is no convenient last statement that can be executed on every iteration. The CONTINUE statement is included in the FORTRAN language to satisfy this requirement. It is a *dummy* statement, since it accomplishes no processing useful to the problem programmer, and is used, whenever needed, for this purpose only.

Whenever the indexing that is required at the end of every DO loop indicates that another iteration is required, control is sent to the beginning of the loop. Thus conditional transfer statements, such as IF, and unconditional transfer statements, such as GO TO, cannot be used as the last statement of a DO loop. There is no restriction on the statements that may be used prior to the last, however, except that the indexing parameters themselves may not be altered. In fact, DO loops may be contained within DO loops, and may have a common last statement. Sometimes it is desired to branch out of a DO loop without completing the specified number of iterations. This is permissible, but branching into a DO loop is not. For correct operation of the DO statement, it must always be the first to which control is sent in the loop.

EXAMPLE 57

A set of 175 integer variables is in memory. Find the largest number in the set.

Solution Procedure

1. Let the variable name for the largest number in the set be LARGE, and set it to a large negative value.
2. Form a loop which compares successive elements of the set to LARGE such that whenever an element larger than LARGE is found, LARGE is reset to the larger value.
3. After each of the 175 members of the set has been tested, LARGE will contain the largest value.

Solution Program

Statement Number	*FORTRAN Statement*
	INTEGER A(175)
	LARGE = −2000000000
	DO 15 I = 1,175
	IF (A(I) − LARGE) 15,15,16
16	LARGE = A(I)
15	CONTINUE
	STOP
	END

12.4.3 Nested DO Loops

FORTRAN allows the construction of DO loops inside of DO loops, as long as one loop is contained entirely within the other. This process is called *nesting* and may be carried to any level. Figure 12.2 illustrates this type of construction. In this example construction, the sequence of events would be as follows:

1. I is set to 1 and the statements between 20 and 30 are executed.
2. J is set to 1 and the statements between 30 and 40 are executed.
3. L is set to 1 and the loop ending with statement 4 is executed four times.
4. The statements between 4 and 10 are executed.
5. J is increased by 1 and the entire process described in steps 2-4 is repeated.
6. J is increased by 1 and 2-4 are repeated a third time.
7. Now the requirements for the J loop are fulfilled and the program goes on to process the statements between 10 and 50.
8. K is set to 1 and that loop is processed seven times.
9. The statements between 12 and 6 are processed.
10. I is increased by 1 and steps 1-10 are repeated.
11. Steps 1-10 are repeated until the requirements of the outermost loop are satisfied.

EXAMPLE 58

Stored in memory are A, a rectangular matrix of three rows and two columns, and B, a rectangular matrix of two rows and two columns. Multiply the matrices A and B to produce C, the three-row, two-column product.

Solution Procedure

Matrix multiplication consists of a number of vector-by-vector multiplications to compute scalar products, as in Example 56. In general, the rule is: the element in row i and column j of the product matrix is the scalar product of row i of the first matrix being multiplied and column j of the second matrix being multiplied. An illustration of this rule follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{pmatrix}; \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}; \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{31} & C_{32} \end{pmatrix}$$

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \\ C_{31} &= A_{31}B_{11} + A_{32}B_{21} \\ C_{32} &= A_{31}B_{12} + A_{32}B_{22} \end{aligned}$$

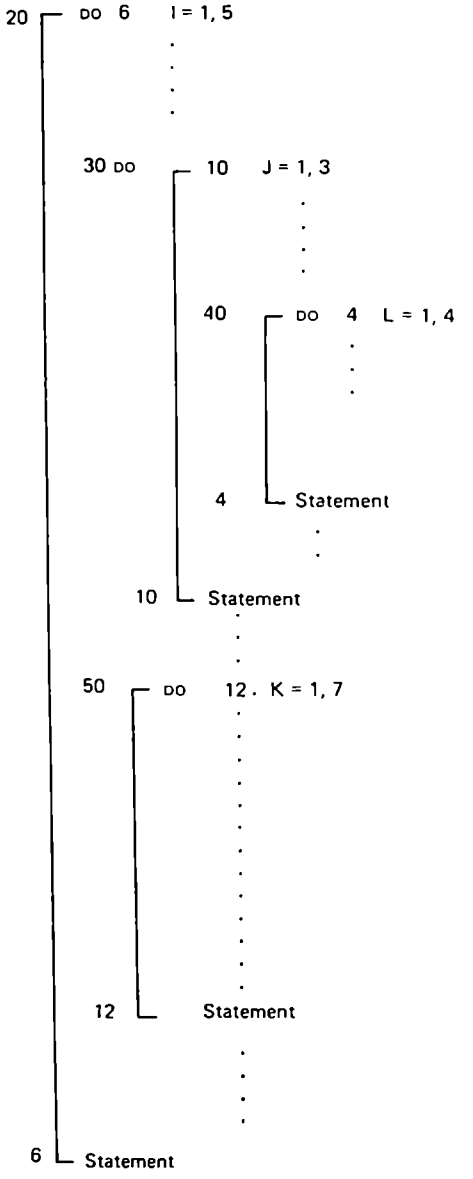


Figure 12.2 Nesting of DO loops

The appropriate solution procedure, therefore, is one that uses the subscript indexing capabilities of DO loops to produce the required scalar products.

Solution Program

Statement Number	*FORTRAN Statement*
	DIMENSION A(3,2), B(2,2), C(3,2)
	DO 5 I = 1,3
	DO 5 J = 1,2

Statement Number	*FORTRAN Statement*
	C(I,J) = 0.
	DO 5 K = 1,2
5	C(I,J) = C(I,J) = A(I,K) * B(K,J)
	STOP
	END

In order to analyze a Solution Program of this complexity, it is often helpful to jot down what happens as each statement is executed so that the logic becomes apparent. Ordinarily it is not necessary to write down more than a part of the program in order to grasp the scheme. In the following, this method of attack is used to shed further light on the behavior of nested DO loops.

Execution Order	*Result*
1	I is set to 1
2	J is set to 1
3	C(1,1) is set to 0
4	K is set to 1
5	C(1,1) = C(1,1) + A(1,1) * B(1,1)
6	K is set to 2
7	C(1,1) = C(1,1) + A(1,2) * B(2,1)
8	J is set to 2
9	C(1,2) is set to 0
10	K is set to 1
11	C(1,2) = C(1,2) − A(1,1) * B(1,2)
12	K is set to 2
13	C(1,2) = C(1,2) + A(1,2) * B(2,2)
14	I is set to 2
15	J is set to 1
16	C(2,1) is set to 0
17	K is set to 1
18	C(2,1) = C(2,1) + A(2,1) * B(1,1)
19	K is set to 2
20	C(2,1) = C(2,1) + A(2,2) * B(2,1)
21	J is set to 2
	Etc.

Note that, as with more straightforward sequences of FORTRAN statements, statements in nested DO loops are executed as they are encountered. This means that if a DO loop nested within another DO loop is encountered, it is indexed through all of its values before either the outer DO loop is indexed or the following statement is executed. This can be made somewhat clearer by rewriting the Solution Program in a slightly longer form as follows:

Alternate Solution Program

Statement Number	*FORTRAN Statement*
	DIMENSION A (3,2), B (2,2), C(3,2)
	DO 5 I = 1,3
	DO 4 J = 1,2

```
Statement Number  FORTRAN Statement
                  C(I,J) = 0.
                  DO 3 K = 1,2
                  C(I,J) = C(I,J) + A(I,K) * B(K,J)
3                 CONTINUE
4                 CONTINUE
5                 CONTINUE
                  STOP
                  END
```

12.4.4 Variable Increments in DO Loops

The basic DO statement may be extended by specifying an increment along with the starting and limiting values:

DO *n* *V*1 = *n*1, *n*2, *n*3

*n*3 is an integer constant or variable whose value is added to *V*1 each time the loop is cycled. When *V*1 exceeds *n*2, cycling is curtailed, as in other cases.

EXAMPLE 59

X is a 5 × 6 array of real numbers. Find the sum (call it PRTSUM) of all elements having even row numbers and odd column numbers.

Solution Program

```
Statement Number  FORTRAN Statement
                  DIMENSION X(5,6)
                  REAL PRTSUM
                  PRTSUM = 0.
                  DO 5 I = 2,4,2
                  DO 4 J = 1,6,2
                  PRTSUM = PRTSUM + X(I,J)
4                 CONTINUE
5                 CONTINUE
                  STOP
                  END
```

The outer loop will start with I (row number) set at 2 and cycle through the inner loop for J (column number) values of 1, 3, and 5. Since 5 is below the limiting value of 6, J will be incremented by 2 and another cycle will be attempted. The new J value (7) will be found to exceed the limiting value, so that series of cycles will be terminated and I will be increased to 4. Then J will be cycled again.

PROBLEMS

1. What are the relative advantages and disadvantages of Assembler Language as compared to that part of FORTRAN thus far encountered?
2. Why is FORTRAN considered a *higher level* language than Assembler Language?
3. What are the kinds of numbers handled by FORTRAN and to what machine language data types do they correspond?
4. What is meant by saying that an arithmetic statement in FORTRAN is not an equation but a replacement operation?
5. In what ways does the DO statement simplify the construction of an iterative loop?

Given the following explicit specifications:

Statement Number	*FORTRAN Statement*
	INTEGER G, H, I, J, K, L(100), N, R
	REAL A, B, C, M(10,20), D
	DOUBLE PRECISION X, Y, Z(3,5,6), W

write FORTRAN programs to accomplish the following:

6. Compute the arithmetic mean of the L-array as a real number, and store it in A.
7. Divide each element in the Z-array by H.
8. Convert I and J to double precision format and store the results in X and Y.
9. Compute 20 terms of $\sin^{-1}x$, for $x^2 < 1$, according to the following formula:

$$\sin^{-1}x = x + \frac{1}{2}\cdot\frac{x^3}{3} + \frac{1}{2}\cdot\frac{3}{4}\cdot\frac{x^5}{5} + \frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\cdot\frac{x^7}{7} + \cdots$$

10. Compute 30 terms of e^x, for all x, according to the following formula:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

11. Compute the standard deviation of the L-array according to the formula:

$$\text{S.D.} = \sqrt{\frac{\Sigma(L_i - \text{Mean})^2}{100}}$$

and store the result as a real number in B. Use the result of Exercise 6.

12. The number of combinations of N things taken R at a time is given by the formula:

$$N\text{C}R = \frac{N!}{R!(N-R)!}$$

Compute this quantity as a double precision number and store it in W.

13. Find the smallest number in the M-array, and store it in C.
14. Find the smallest positive number in the M-array, and store it in D.
15. Repeat Exercise 9, storing the smallest positive number of the M-array in D, and storing its row number and column number in G and H, respectively.

Chapter 13

FORTRAN Input/Output

In keeping with the relative simplicity of the arithmetic and logical capabilities of FORTRAN, its input/output statements number but six, and half of these do all but a small fraction of the work. Five of the input/output statements (READ, WRITE, END FILE, REWIND, and BACKSPACE) are executable statements and refer to input/output devices; the sixth statement (FORMAT) is a specification statement used in conjunction with READ and WRITE to designate the format of the data.

References to input/output devices by the five executable statements are made in terms of small integers. The correspondence between the devices available in any System/360 installation and the numbers used to designate such devices is not necessarily fixed. This correspondence may be set at a particular installation by assembling or reassembling the FORTRAN compiler using appropriate control cards. It may also be made at the time of program compilation or at the time of program execution, also by means of control cards. As distributed by IBM, however, the FORTRAN programming system uses a standard set of device reference numbers, and it is assumed that this set is the one most commonly used. At any rate, this is the set used here to illustrate and discuss the programming of input/output statements in System/360 FORTRAN. The table shows the devices and their corresponding reference numbers as used in subsequent examples.

Input/Output Device	*Device Reference Number*
Card Reader (2540)	5
Card Punch (2540)	7
Printer (1403)	6
Tape Drive (2400)	1
Tape Drive (2400)	2

As used in programming, device reference numbers may be expressed either as unsigned integer constants or as integer variables.

13.1 BASIC OUTPUT OPERATIONS

13.1.1 The WRITE Statement

The WRITE statement handles all output in FORTRAN. We shall consider this statement first because by generating data internally we can write complete programs, including the display of results. Its basic form is

WRITE (*n*) *var*1, *var*2, etc.

n is the device reference number and *var*1, *var*2, etc. represents a list of variables which will be written in the order specified. This most basic version will produce output in binary mode, and therefore is not suitable for direct display. For such purposes we use the form

WRITE (*n*,*m*) *var*1, *var*2, ...

where *m* refers to the number of a statement elsewhere in the program which describes the exact appearance of the display.

13.1.2 The FORMAT Statement

Description of output is handled by the FORMAT statement. This is a nonexecutable statement and may be placed anywhere in the program. A FORMAT statement must be accompanied by a statement number since there will be a reference to it in some WRITE statement (or READ statement, as we shall see later). The form is

m FORMAT (*spec*1, *spec*2, ...)

*spec*1, *spec*2, refer to a list of format specifications which describe the respective appearances of the items in the WRITE list. A particular FORMAT statement may be cited by several different WRITE statements in a program, as long as the output listed by them is designed to have the same layout.

13.1.2.1 The I Specification

If integer output is to be produced, each item is described by a format specification of the type

I(*w*)

where *w* is an integer constant defining the number of digits to be used (and therefore the number of spaces to be reserved) to express a result. If the item's value requires $< w$ digits, the value is displayed as right-justified, with the surplus high-order digits being filled with blanks. If the value exceeds *w* digits, the appropriate number of leftmost digits is truncated.

EXAMPLE 60

Add the integers from 1 through 1,000; then punch the sum (call it N), and the sum of all the odd numbers (call it NODD), allowing six digits for each value.

Solution Procedure

1. This program requires no input. Use a DO loop to compute the sums.
2. Use the WRITE statement to punch the sum into a card and the FORMAT statement to specify the columns to be used.

Solution Program

```
   Statement Number   FORTRAN Statement
                      NODD = 0
                      N = 0
                      DO 8 I = 1, 1000, 2
                      NODD = NODD + I
8                     N = N + I + I + 1:
                      WRITE (7,10) N, NODD
10                    FORMAT (I6,I6)
                      STOP
                      END
```

As a result of the WRITE statement, the output card's contents will be as shown:

Columns	*Item*	*Value*
1-6	N	500500
7-12	NODD	250000

13.1.2.2 The x Specification

We can arrange our output layout so that a designated number of spaces is skipped between successive items in our WRITE list. This is done with the specification

*w*X

where w is an integer constant indicating the number spaces to be skipped. Going back to the previous example, if we wanted N to appear in columns 5–10 and NODD is to be punched in 13–18, our new FORMAT statement would be

```
10  FORMAT (4X, I6, 2X,I6)
```

13.1.2.3 The F Specification

Formats for real numbers are defined in one of two ways. The more conventional form is described by the F specification, which is written as

F(w.d)

where w specifies the total number of digits and d tells how many of those digits are to appear to the right of the decimal place. The value given for w must include one space for the decimal point itself and one for the sign. There must be room for at least one digit to the left of the decimal point. Some examples of the F specification are shown in Table 13.1.

EXAMPLE 61

Repeat Example 60 but this time punch N in columns 1–6, $\sqrt{N}$ (call it S) in the form XXX.XXX in columns 10–16 and NODD in 21–26.

Solution Program

Statement Number	*FORTRAN Statement*
	NODD = 0
	N = 0
	REAL S
	DO 8 I = 1, 1000, 2
	NODD = NODD + 1
8	N = N + 2*I + 1
	S = N
	S = SQRT (S)
	WRITE (7,10) N, S, NODD
10	FORMAT (I6, 3X, F7.3, 4X, I6)
	STOP
	END

TABLE 13.1 USE OF THE F SPECIFICATION FOR OUTPUT

Internal Value	Specification	Appearance
204	F3.0	204
204	F6.2	204.00
−17.32	F7.3	−17.320
−17.32	F5.2	17.32
−.064	F6.3	−0.064

The statement S = N gives us the value of N as a real number for square root extraction.

13.1.2.4
Carriage Control on the Line Printer

When output is written on the printer (device 6 for our purposes) the first specification in the associated FORMAT statement is used by FORTRAN as carriage-control information. This takes the form of a single character enclosed by apostrophes. Four types of instructions may be given:

Carriage-Control Character	*Indicated Action*
'$\flat$'	Go to the next line before printing.
'0'	Advance two lines before printing.
'+'	Print on the current line.
'1'	Start a new page.

13.2
BASIC INPUT OPERATIONS

13.2.1
The Basic READ Statement

The READ statement handles FORTRAN's input activities. Its general form

READ (*n*,*m*) *var*1, *var*2, ...

is analogous to the WRITE statement. (A binary version also exists.) Note that the variables listed in a READ statement need not be defined in previous specification statements (unless they are arrays). If no formal specifications are given, FORTRAN will use the naming rules (see Chapter 12) to assign the proper storage. Thus when variables are to be treated as double-precision, they must always be explicitly declared.

The I, X, and F format specifications operate with input in the same way as described for output. We shall usually assume that our input is on punched cards (device 5). When this is the case, FORTRAN operates on the basis that each READ statement operates on a new card, starting with column 1.

EXAMPLE 62

One thousand integers in the range $-9{,}999 \leq I \leq 9{,}999$ are punched in cards, one per card. Each is in columns 1–5 with the minus sign preceding those that are negative. Find the range (difference between the largest and the smallest) and the arithmetic mean of these integers. Print the range as an integer and the mean as a real number to two decimal places in print position 1–5 and 11–18, respectively.

Solution Procedure

1. Set to initial values a largest number variable, a smallest number variable, and a sum variable.
2. Write a loop which, on each iteration, reads a datum from a card, replaces either the smallest or largest number variable if appropriate, and updates the sum variable.
3. After all the data have been read, compute the mean and the range.
4. Print the results.

Solution Program

```
Statement Number   FORTRAN Statement
                   INTEGER SMALL, SUM, RANGE
                   REAL MEAN
                   SMALL = 10000
                   LARGE = -10000
                   SUM = 0
                   DO 20 N = 1, 1000
                   READ (5,4) I
                   IF (I - SMALL) 10, 11, 11
10                 SMALL = I
11                 IF (I - LARGE) 20, 20, 21
21                 LARGE = I
20                 SUM = SUM + I
                   RANGE = LARGE - SMALL
                   MEAN = SUM/1000.
                   WRITE (6,30) RANGE, MEAN
                   STOP
4                  FORMAT (I5)
30                 FORMAT ('1', I5, 5X, F8.2)
                   END
```

Division of SUM by 1000. (real) rather than 1000 (integer) ensures the accuracy of the result. Had we specified 1000, the resulting integer/integer division would have produced an integer quotient (truncated) which would then have been converted to real.

13.2.2 Handling Input Arrays

The READ statement can be expanded to handle arrays or parts of arrays by including specifications very similar to those in a DO loop. If ARAYV represents the name of a dimensioned array variable, the expanded READ statement takes the form

READ (n, m) (ARAYV($v1$), $V1 = n1, n2$)

where $v1$, $n1$ and $n2$ have the same meanings as in a DO statement. As a result, the next ($n2 - n1 + 1$) items would be read in and stored in ARAYV($n1$), ARAYV($n1+1$), . . . ARAYV ($n2$). As with the DO-loop, an

increment other than 1 may be explicitly indicated, in which case the general form would be

READ (*n*,*m*) (ARAYV(*V*1), *V*1 = *n*1, *n*2, *n*3)

EXAMPLE 63

Repeat the previous example with the input values being punched ten per card, in columns 1-5, 6-10, and so forth through 46-50.

Solution Procedure

The procedure is primarily as before. Two loops are used, however, instead of one. Whereas in the previous example there were 1,000 cards to be read and 1,000 data values to be processed, calling for the use of a single loop, here there are 100 cards to be read and the same 1,000 data to be processed.

The ten variables in each card may be given separate names, but if this is done, the processing statements must be repeated for each. To facilitate using a second loop to process the data on each card, a 10-element array is defined so that subscripts may be used.

Solution Program

Statement Number	*FORTRAN Statement*
	INTEGER SMALL, SUM, RANGE, I(10)
	REAL MEAN
	SMALL = 10000
	LARGE = −10000
	SUM = 0
	DO 20 N = 1,100
	READ (5,4) (I(L), L = 1,10)
	DO 20 M = 1,10
	IF (I(M) − SMALL) 10, 11, 11
10	SMALL = I(M)
11	IF (I(M) − LARGE) 20, 20, 21
21	LARGE = I(M)
20	SUM = SUM + I(M)
	RANGE = LARGE − SMALL
	MEAN = SUM/1000.
	WRITE (6,30) RANGE, MEAN
	STOP
4	FORMAT (10I5)
30	FORMAT ('1', I5, 5X, F8.2)
	END

Note that statement 4, the input FORMAT statement, uses a shorthand form in which consecutive identical specifications are replaced by a multipler. Thus (10I5) is equivalent to (I5, I5, I5, I5, I5, I5, I5, I5, I5, I5).

Multidimensional input arrays are handled the same way. For example, the statement

READ (5,12) ((Y(I,J), J = 1,2), I = 1,3)

will read the next six input values into Y(1,1), Y(1,2), Y(2,1), Y(2,2), Y(3,1), and Y(3,2), respectively using the format given in statement 12. Had we written

```
READ (5,12) ((Y(I,J), I = 1,3), J = 1,2)
```

the order for storage would have been Y(1,1), Y(2,1), Y(3,1), Y(1,2), Y(2,2), Y(3,2).

13.2.3 The Specification for Input

Most real-input data are punched without decimal points. Instead, its location is implied in each data item. Consequently, when an input format is specified, FORTRAN will handle the necessary adjustment automatically. Thus if a six-digit value is read in with an F6.4 specification it will be treated as XX.XXXX. If an input value does have an actual decimal point punched in it and its placement does not agree with that implied in the F specification, the actual datum overrides the format specification, as would be expected.

EXAMPLE 64

Each card in a set of 1,000 has three factors punched in it. A is punched in columns 1-9; B is punched in columns 15-21; and C is punched in columns 33-39. Some of the numbers are punched with the decimal point, and some without. For those numbers punched without a decimal point, it is understood to be between the first and second digit from the low-order end of the field. Print a line for each card, with A, B, C, and A + B + C in position 1-10, 11-20, 21-30, and 31-40 of the output form, respectively, and with one digit to the right of the decimal point printed for each.

Solution Program

Statement Number	*FORTRAN Statement*
	DO 1 I = 1,1000
	READ (5,8) A, B, C
	SUM = A + B + C
1	WRITE (6,9) A, B, C, SUM
	STOP
8	FORMAT (F9.1, 5X, F7.1, 11X, F7.1)
9	FORMAT ('b', 4F10.1)
	END

13.3 ADDITIONAL FORMAT POSSIBILITIES

Now that we have established FORTRAN's basic I/O capabilities we can go back and pick up some additional features which extend the programmer's abilities to arrange more elaborate output formats and accept more complex input arrangements.

13.3.1 The H Specification

The specification of output displays need not be restricted to values stored in memory under assigned symbolic names. Such output may be supplemented (or replaced) by literal values. These are defined in the FORMAT statement itself as part of the specification list.

Literal values are specified by preceding the desired string of characters with the specification

*w*H

when *w* is an integer constant equal to the number of characters in the literal value. *w* must agree with the length of the actual string. Remember that every character (including blanks, commas, and so forth) counts in the total. Thus, if we wish to specify the literal "X IS EQUAL TO" (excluding the quote marks) we would write

13HX IS EQUAL TO

To show the H specification's usage in context, let us say we have a variable named X and we wish to print a line in which the first 25 spaces are blank, followed by the above message, followed by three blanks and, finally, X itself in the form F8.3. Our statement would read

```
n   FORMAT (25X, 13HX IS EQUAL TO, 3X, F8.3)
```

or

```
n   FORMAT (25X, 16HX IS EQUAL TO  , F8.3)
```

EXAMPLE 65

A set of 1,000 real numbers is punched 10 per card in F8.0 format. Find the median (the number above which half the values lie) and print it, preceded by the message "THE MEDIAN OF THIS ARRAY IS."

Solution Procedure

1. Store all the numbers.
2. Sort the numbers in descending order.
3. Compute the median as the average of the 500th and 501st elements in the sorted array.
4. Print the literal message, followed by the median in F10.1 format.

Solution Program

Statement Number	*FORTRAN Statement*
	REAL MEDIAN, A(1000)
	READ (5,8) A
8	FORMAT (10F8.0)

```
Statement Number   FORTRAN Statement
15                 TEST = 0
                   DO 20 I = 1,999
                   IF (A(I) - A(I + 1)) 4, 20, 20
4                  TEST = A(I)
                   A(I) = A(I + 1)
                   A(I + 1) = TEST
                   TEST = 1.
20                 CONTINUE
                   IF (TEST) 15, 16, 15
16                 MEDIAN = (A(500) + A(501))/2.
                   WRITE (6,30) MEDIAN
30                 FORMAT ('1', 28H THE MEDIAN OF THIS
                   ARRAY IS, F10.1)
                   STOP
                   END
```

Note that there is no reference to the literal message in the WRITE statement.

An alternative method for specifying literals is enclosing the desired character string in apostrophes. This is somewhat simpler than the H specification since no character count is required. Thus statement 30 in the previous example could have been written

```
30  FORMAT ('1', 'THE MEDIAN OF THIS ARRAY IS', F10.1)
```

It is not always advantageous to use this simplified form. There are many occasions when layout considerations require a careful character count.

The H specification can also be used for input. For example, if an input card has SCORE IS 279 in columns 1-14, the sequence

```
      READ (5,10) X
10    FORMAT (10X,8HHEADINGS, 3X, I3)
            .
            .
            .
      WRITE (6,10) I
```

will produce

```
SCORE IS  279
```

13.3.2 The E and D Specifications

Input/output of numbers expressed in exponential form is handled by the E and D specifications. The general form is

$$\begin{cases} \text{E} \\ \text{D} \end{cases} w.d$$

where w and d have the same meanings as in the F specification.

When used for input, this specification indicates the form in which the

input is to be stored. As such, it allows wide flexibility for expression. Thus 8.18E2 may be punched 8.18E2, 8.18E+2, 8.18E+02, 8.18E02, 8.18+2, or 8.18+02. When a D is punched instead of an E, the corresponding variable name in the READ statement must be declared with DOUBLE PRECISION in a specification statement.

For output, the E or D specification must allow for a sufficient number of digits (w) to accommodate the value's sign, one digit to the left of the decimal point, the decimal point, d digits for the fractional portion, the actual E or D, an exponent sign, and the two-digit exponent. This is shown for several sample values.

Stored Value	*Format Specification*	*Appearance As Output*
174.	E11.4	b0.1740Eb03
174	D10.4	0.1740Db03
−.0061	E12.5	−0.61000E−02
−.0061	D10.4	0.6100D−02
−614.0627	E12.4	b−0.6140Eb03

13.3.3 The / Specification

A slash in a FORMAT statement signals FORTRAN to skip to the next record. Slashes may appear anywhere in a FORMAT statement (except in the middle of a specification). When used for input, the slash forces FORTRAN to go to the beginning of a new card. Thus, the sequence

```
      READ (5,10) X, Y, Z, W
10    FORMAT (12X, F5.2, 10X, F5.0/6X, F3.10,/F10.5)
```

will result in X and Y being obtained respectively from columns 13–17 and 28–32 of the first card, Z from columns 7–9 of the next card and W from columns 1–10 of the card after that. Had we written

```
10    FORMAT (12X, F5.2, 10X, F5.0//6X, F3.0/F10.5)
```

instead, an entire card would have been read in and ignored after Y was picked up, so that Z would be read from the third card, and W from a fourth.

On output, the slash operates similarly, causing the printer to go to the beginning of a new line. A slash at the end of an output FORMAT statement will cause a line to be skipped immediately after printing.

13.3.4 The T Specification

The specification

Tw

may be used instead of X to cause w spaces to be skipped before the next

item is read or written. It is called T because it acts like the tab key of a typewriter. Thus the two statements given below

```
FORMAT (X9, F9.2, X12, I4, X3, 2F6.2, X4, I6)
FORMAT (T10, F9.2, T31, I4, T38, 2F6.2, T54, I6)
```

produce identical formats.

EXAMPLE 66

A set of data is punched six per card in columns 1–12, 13–24, 25–46, 37–48, 49–60, and 60–72. Because of the magnitudes involved, these real data are punched using the E exponent notation (4.935,000,000 is punched as 4.935E9, and so on). The number of data items is less than 1000 but otherwise unknown. Following the last item and punched as if it were a piece of data, is the quantity 9.999E-20. Divide the range of this set of data into 20 equal class intervals, and form a frequency distribution. Print a column showing the bounds of each class interval and the associated frequency.

Solution Procedure

1. Read the data and store them consecutively, until the quantity 9.999E-20 is reached. While reading, find the largest and the smallest datum in the set.
2. Compute the range and the class interval size.
3. Write a loop to inspect each datum and add 1 to the frequency of the appropriate class interval. (An alternate way of doing this is to sort the data, then count how many there are between the upper and the lower limits of each class interval.)
4. Print the results, showing limits and frequency for each class interval.

Solution Program

Statement Number	*FORTRAN Statement*
	DIMENSION N(20), A(1000), B(6)
	BIG = 9.999E−20
	SMALL = 9.999E50
	I = 0
5	READ (5,10) B
	DO 41 J = 1,6
	IF (B(J) − 9.999E−20) 20, 21, 20
20	IF (B(J) − BIG) 30, 30, 31
31	BIG = B(J)
30	IF (B(J) − SMALL) 40, 41, 41
40	SMALL = B(J)
41	A (I + J) = B(J)
	I = I + 6
	GO TO 5
21	K = I + J − 1
	RANGE = BIG − SMALL
	CLASS = RANGE/20.

```
Statement Number   FORTRAN Statement
                   DO 4 I = 1,20
4                  N(I) = 0
                   DO 50 I = 1,K
                   DO 1 J = 1,20
                   IF (SMALL + J*CLASS - A(I)) 1, 2, 2
2                  N(21-J) = N(21-J) + 1
                   GO TO 50
1                  CONTINUE
50                 CONTINUE
                   WRITE (6,15) RANGE, CLASS
                   WRITE (6,16)
                   DO 80 I = 1, 20
                   TOP = BIG - (I-1)*CLASS
                   BOTTOM = BIG - I*CLASS
80                 WRITE (6,17) TOP, BOTTOM, N(I)
                   STOP
10                 FORMAT(6E12.3)
15                 FORMAT ('1', 13HTHE RANGE IS, E12.3, 26H AND THE IN-
                   TERVAL SIZE IS, E12.3)
16                 FORMAT (///'b', 8X, 20HCLASS INTERVAL RANGE, 14X,
                   9HFREQUENCY/)
17                 FORMAT ('b', 4X, E12.3, 4H TO, E12.3, 14X, I3)
                   END
```

13.3.5
The P Specification

The P format specification may be used to modify the order of magnitude (in other words, the scale) of real input or output data. It is most commonly used with data described by the F specification and takes the form

*n*P*Fw.d*

where *n*, the *scaling factor*, has the following effect:

$$\text{External value} = \text{Internal value} \times 10^n$$

As an example, let us say that a particular input value is punched as 3714.68. The effect of scaling on its internal value is as follows:

Format Specification	*Stored As*
F7.2	3714.68
1PF7.2	371.468
2PF7.2	37.1468
4PF7.2	.371468
−3PF7.2	3714680.

The same relationships hold for output. This is shown for several example scaling factors for a stored value of 3714.68:

F7.2	3714.68
2PF7.2	371468.
−2PF7.2	37.1468
0PF7.2	3714.68

EXAMPLE 67

A two-dimensional array with no more than 30 rows and 30 columns is punched in cards in 6F10.3 format. The elements are consecutive and punched row by row. They are preceded by a card in which the specific numbers of rows and columns are punched in 2I2 format. Read the matrix into memory, at the same time-scaling up the elements by a factor of 10. Find the column whose elements have the greatest sum and write the elements of that column on magnetic tape in external (zoned-decimal) format. Find the column whose elements have the smallest sum and write the elements of that column on another magnetic tape in internal (binary) format.

Solution Program

Statement Number *FORTRAN Statement*

```
      DIMENSION X(30, 30), S(30)
C     I IS THE NO. OF ROWS. J IS THE NO. OF COLUMNS. THESE VAL-
C     UES WILL BE READ IN TO DETERMINE THE EXTENT OF SUB-
C     SEQUENT INPUT.
      READ (5,10) I, J, ((X(K, L), L = 1, J), K = 1, I)
10    FORMAT (2I2/(-1P6F10.3))
C     THE ENCLOSURE OF -1P6F10.3 IN PARENTHESES CAUSES FOR-
C     TRAN TO CONTINUE USING THAT FORMAT OVER AND OVER
C     TILL INPUT IS COMPLETED. THAT IS THE FIRST CARD WILL BE
C     HANDLED AS 2I2 FOLLOWED BY A NUMBER OF CARDS (DETER-
C     MINED BY I AND J) HANDLED BY -1P6F10.3
      DO 8 N = 1, J
8     S(N) = 0.
      DO 15 N = 1, J
      DO 15 M = 1, I
15    S(N) = S(N) + X(M,N)
      BIG = 1.0E - 20
      SMALL = 1.0E+20
      DO 23 N = 1, J
      IF (S(N) = BIG) 20, 20, 21
21    BIG = S(N)
      LB = N
20    IF (S(N) = SMALL) 22, 23, 23
```

```
Statement Number   FORTRAN Statement
22                 SMALL = S(N)
                   LS = N
23                 CONTINUE
                   WRITE (1, 18) (X(K, LB) K = 1, I)
18                 FORMAT (30F10.3)
                   WRITE (2) (X(K,LS), K = 1, I)
C  NO FORMAT NEEDED FOR THE ABOVE BECAUSE WE ARE WRIT-
C  ING IN BINARY
                   STOP
                   END
```

13.3.6 The A Specification

Thus far we have dealt with I/O of numerical variables and, to a small extent, with literals. FORTRAN provides additional features for handling nonnumeric (i.e., character) data. When data are to be stored or displayed as characters (one to a byte) they are described by the specification

```
An
```

where *n*, an integer constant, indicates the number of characters to be transmitted.

13.4 TAPE OPERATIONS

These additional statements are available to aid in handling the transmission of data to and from magnetic tape. The REWIND statement, whose general form is

```
REWIND m
```

causes the tape mounted on the device whose code number is *m* to be repositioned so that its first record is available for reading (or modification by an appropriate WRITE instruction).

The END FILE statement, which is written

```
END FILE m
```

places a special indicator (*tape mark*) after the last record that was written. This defines the end of a particular collection of data. (There may be several data collections, and therefore several tape marks, on a single reel of tape.)

The BACKSPACE statement follows the same form,

```
BACKSPACE m
```

and causes the designated tape to be positioned one record closer to its beginning.

EXAMPLE 68

A file of 2,500 data records has been punched in a deck of cards. There is one record per card, consisting of a 12-digit alphabetic field and six 5-digit integer fields. Transfer this file of data to magnetic tape for later use in retrieving the stored data. This newly created tape file is to be matched against a set of 25 cards, each containing characters in columns 1-12. When the alphabetic portion of a tape record is the same as the contents of one of the cards, punch that tape record on a new card.

Solution Program

Statement Number	*FORTRAN Statement*
	DIMENSION ALF (75), ALPHA (3)
	REWIND 1
	DO 20 INDEX = 1, 2500
	READ (5, 10) ALPHA, I, J, K, L, M, N
10	FORMAT (3A4, 6I5)
20	WRITE (1) ALPHA, I, J, K, L, M, N
	END FILE 1
	REWIND 1
	READ (5,11) (ALF(K), K = 1, 75)
11	FORMAT (3A4)
	DO 30 INDEX = 1, 2500
	DO 30 INDEX = 1, 25
	INDIC = 3*INDEX − 2
	IF (ALPHA (1) − ALF(INDIC)) 30, 31, 30
31	IF (ALPHA (2) − ALF(INDIC+1)) 30, 32, 30
32	IF (ALPHA (3) − ALF(INDIC+2)) 30, 33, 30
33	WRITE (7,10) ALPHA, I, J, K, L, M, N
30	CONTINUE
	STOP
	END

Note that the alphameric field (12 bytes) is handled as three four-byte words here. Had we specified ALPHA or ALF as double-precision arrays, we would have reserved a corresponding number of doublewords. Because of the way FORTRAN is constructed we cannot write statement 10 as

```
10  FORMAT (A12, 6I5)
```

Instead, we must read alphameric data into fullwords or doublewords. Of course, we could have placed each byte in its own word (thus tying up four times as much storage) by declaring ALPHA, for example as ALPHA (12) and reading the data with

```
10  FORMAT (12A1, 6I5)
```

PROBLEMS

1. What is the reason that the specification statements REAL, INTEGER, DIMENSION, and DOUBLE PRECISION must be written before the first executable statement of a program, while the specification statement FORMAT may be placed anywhere?
2. When are READ and WRITE statements used without a FORMAT specification, and what is the advantage of doing so?
3. What is the difference in the output produced by the following sequences of instructions:

   (a) *First Sequence*

```
      DO 10 I = 1, K
10    WRITE (2, 15) (A(I,J), J = 1, L)
15    FORMAT (6E12.5)
```

   (b) *Second Sequence*

```
      WRITE (2, 15) ((A(I, J), J = 1, L), I = 1, K)
15    FORMAT (6E12.5)
```

   (c) *Third Sequence*

```
      DO 10 I = 1, K
      DO 10 J = 1, L
10    WRITE (2, 15)  A (I,J)
15    FORMAT (6E12.5)
```

4. How many cards are read by each of the following sequences of instructions?

   (a) *First Sequence*

```
      READ (1, 5) A, B, C
5     FORMAT (6D12.4)
```

   (b) *Second Sequence*

```
      READ (1, 5) A, B, C
5     FORMAT (E8.1)
```

   (c) *Third Sequence*

```
      READ (1, 5) (A(I), I = 1, 20, 3)
5     FORMAT (2E10.3)
```

   (d) *Fourth Sequence*

```
      DO 8 I = 1, 20, 3
      READ (1, 5) (A(I,J), J = 5, 14, 4)
5     FORMAT (3F8.4)
```

(e) *Fifth Sequence*

```
      READ (1, 5) ((A(I,J), I = 1, 20, 3), J = 5, 14, 4)
5     FORMAT (3F8.4)
```

5. Following are two programs for computing the arithmetic mean and standard deviation of 100 numbers punched 5 per card. Both programs use the same formulas:

$$\text{S.D.} = \frac{1}{N}\sqrt{N\Sigma X^2 - (\Sigma X)^2} \qquad \text{Mean} = \frac{\Sigma X}{N}$$

in which the X's are the values of the numbers and N is their number (in this case 100). Find and discuss the major difference between the programs.

(a) *First Program*

Statement Number	*FORTRAN Statement*
	REAL X(100), MEAN
	READ (1, 5) (X(I), I = 1, 100)
5	FORMAT (5F8.2)
	SUMX = 0.
	SUMXX = 0.
	DO 8 I = 1, 100
	SUMX = SUMX + X(I)
8	SUMXX = SUMXX + X(I)**2
	MEAN = SUMX/100.
	SD = (SQRT(100.*SUMXX − SUMX**2))/100.
4	WRITE (2,4) MEAN, SD
	FORMAT ('MEAN IS', F8.2, 4X,
	'S.D. IS', F8.2)
	STOP
	END

(b) *Second Program*

Statement Number	*FORTRAN Statement*
	REAL X(5), MEAN
	SUMX = 0.
	SUMXX = 0.
	DO 8 I = 1, 20
	READ (1,5) (X(K), K = 1,5)
5	FORMAT (5F8.2)
	DO 8 J = 1, 5
	SUMX = SUMX + X(J)
8	SUMXX = SUMXX + (X(J))**2
	MEAN = SUMX/100.
	SD = (SQRT(100.*SUMXX − SUMX**2))/100.
	WRITE (2, 4) MEAN, SD

Statement Number	*FORTRAN Statement*
4	FORMAT ('MEAN IS', F8.2, 4X, 'S.D. IS', F8.2)
	STOP
	END

6. A set of 200 integers is punched 4 per card. Write a program to read these data into memory, then to punch them out 5 per card.
7. Write a program to read the data of the previous exercises to memory, then to punch them out 6 per card and in the reverse order.
8. Write a program to read the data of the previous exercises and punch them out 3 per card, without specifying an array of more than 8 elements.
9. A set of N integers is punched M per card. Write a program to read these data and punch them out 1 per card, without specifying an array of more than $2*M$ or $2*L$ elements, whichever is the greater.
10. Write a program to create a printed table of the integers from 1 through 100 and their squares, cubes, and square roots. Include a title and column headings.
11. Write a program that will test whether or not the first three columns of a data card contain the word YES. If they do, print DATA CARD CONTAINS YES; if they do not, print DATA CARD DOES NOT CONTAIN YES.
12. A square matrix of N rows and N columns is recorded on magnetic tape in internal format. The first quantity is N, and it is followed by the matrix elements in column by column order. N is less than or equal to 20. Multiply this matrix by its transpose (the same matrix with rows and columns interchanged), and print the product in row by row order.
13. Write a program that will test whether or not any of the characters punched in a card is a letter of the alphabet.
14. As X varies from -10.0 to $+10.0$ in steps of 0.1, $X**2$ varies from 100 to 0 and back to 100 again. Write a program to round each value of X^2 to the nearest integral value and to print a decimal point in the print position corresponding to each of these integers. This will produce the graph of $Y = X**2$ as X is varied between these limits. [*Hint:* Use a 101-character array of blanks, and insert the decimal point as calculated. Then print the whole array.]
15. Write a program to solve the quadratic equation. $A*X**2 + B*X + C = 0$, such that complex roots are printed in the form $a + bi$. For example, a complex root whose real part is 1.003 and whose imaginary part is -4.232 should be printed as 1.003 -4.2321.

Chapter 14

Additional FORTRAN Features

Although the features discussed in the last two chapters permit the user to prepare complete FORTRAN programs over a wide range of complexities, they constitute only a basic subset of the language's capabilities. In this chapter we shall take a brief look at some additional features which facilitate the organization of programs and extend the repertoire of operations.

14.1 SUBPROGRAMS

FORTRAN provides a set of relatively straightforward mechanisms for setting up and using subprograms. Depending on their complexity and anticipated usage, they may be organized in one of three ways.

14.1.1 Statement Functions

A *statement function* is defined by a single arithmetic statement in which a particular name is assigned to a certain set of operations. Once this is done, the programmer may implement those operations anywhere in his program merely by referring to that name. This is handled in exactly the same way as one of FORTRAN's built-in functions.

A statement function is defined by a statement of the general form

name (argument, argument, ...) = arithmetic expression

The items in parentheses are variable names corresponding to those used in the arithmetic expression. These are called *arguments* and their number depends, of course, on the nature of the function. It should be understood that these arguments do not represent actual variables for which storage has been allocated. Instead, they are *dummy* names set up for reference purposes. When the programmer uses the function he specifies the variables of interest.

As an illustration, let us suppose that we have use for a value obtained by adding the square of one variable to twice the square of another and taking the square root of that sum. Assigning the name VECT to this function, we can define it by the statement

```
VECT(X,Y) = SQRT(X**2 + 2.*Y**2)
```

This statement is placed ahead of other executable statements in the program. Then at some later point in the program if we wrote

```
Z = 3. + VECT(A,B)
```

a value equal to $3 + \sqrt{A^2 + 2B}$ would be placed in Z. Had we written

```
Z = 3. + VECT(B,A)
```

the value would be calculated as $3 + \sqrt{B^2 + 2A}$.

Statement functions may be built with as many as 15 arguments, all of which must be nonsubscripted variables. The type of arguments required and the type of value produced are defined by the initial letter of the function's name. As is the case with variable names, the type can be explicitly specified. Thus we could have written

```
REAL VECT(X,Y) = SQRT(X**2 + 2.*Y**2)
```

EXAMPLE 69

Find a root, correct to four decimal places, of the equation

$$2x^5 - 7x^4 + 3x^3 - 4x^2 - 4x + 3 = 0$$

Solution Procedure

We shall use Newton's method for determining a root. This procedure starts with an approximation for x and uses it to obtain the next, more accurate one. The process is repeated until a particular value differs from the preceding approximation by no more than a certain amount. If x_n is the nth approximation, then

$$x_{n+1} = x_n - f'(x_n)/f'(x_n)$$

where $f(x_n)$ is $2x_n^5 - 7x_n^4 + 3x_n^3 - 4x_n^2 - 4x_n + 3$ and $f'(x_n)$ is its first

derivative. This derivative value can be approximated by

$$\frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

If x_2 and x_1 are taken very close to each other, this ratio comes very close to the first derivative value. By using x_n as x_2 and $x_n - x$ as x_1, we can rewrite the expression for the derivative value as

$$\frac{f(x_n) - f(x_n - \Delta x)}{\Delta x}$$

and then

$$x_{n+1} + x_n - \frac{f(x_n)}{\dfrac{f(x_n) - f(x_n - \Delta x)}{\Delta x}} = x_n - \frac{f(x)\ \Delta x}{f(x_n) - f(x_n - \Delta x)}$$

For our program we shall use 1×10^{-8} as Δx. The polynomial will be evaluated using Horner's method, which presents a more efficient scheme. In general, with a fifth-degree polynomial

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5$$

Horner's method uses the form

$$f(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + x(a_4 + x(a_5)))))$$

In our specific case this is

$$f(x) = 3 + x(-4 + x(-4 + x(3 + x(-7 + x(2)))))$$

We shall use a statement function named POLY to obtain this value each time it is needed. 0.0001 will be used as the maximum deviation between successive x approximations.

Solution Program

Statement Number	*FORTRAN Statement*
	POLY(A) = 3.+A*(−4.+A*(−4.+A*(3.+A*(−7.+A*(2.)))))
	XN = 0.
	H = 1.0E−8
	H2 = H/2.
1	U = XN
	V = XN−H2
	P = POLY(U)
	Q = POLY(V)
	Y = POLY(XN)
	XN1 = XN−(Y*H)/(P−Q)
	IF(ABS(XN1 − XN)−.0001)10,11,11
11	XN = XN1
	GO TO 1
10	WRITE(6,15)XN1

Statement Number	*FORTRAN Statement*
15	FORMAT ('b','POLYNOMIAL ROOT IS',E12.4)
	STOP
	END

14.1.2 Function Subprograms

When it is inconvenient (or impossible) to express a desired sequence of operations in a single statement, such operations can be organized as a *function subprogram.* Such routines are organizational entities in themselves. One can be compiled separately and added to any program which might require that particular set of calculations.

Each function subprogram ends with an END statement and begins with a statement having the form

FUNCTION *name* (*argument,argument,* . . . .)

The desired operations are then specified and are followed by a statement which merely says

RETURN

This apparently simple statement is backed by a complex mechanism which automatically arranges for the storage of return branch destinations and executes the actual branch back to the calling program. As with statement functions, the type of result returned is implied by the function's name or may be specified ahead of the keyword FUNCTION.

EXAMPLE 70

Write a program which reads in sets of values A, B, C, D, E, AND F and, for each one, calculates a value Y as

$$Y = 2X + \sqrt{X}$$

where X is a solution for the equation

$$AX^5 + BX^4 + CX^3 + DX^2 + EX + F = 0$$

Each set of values is punched on a card in F3.1 format in columns 1–18. Results are to be printed one to a line in E12.4 format, along with the input values using the following column headings:

A B C D E F Y

The last set of data is followed by a dummy set of zeros.

Solution Procedure

We shall use the program in the previous example as a function subprogram to illustrate its organization and relation to the main program.

Solution Program

```
Statement
Number    FORTRAN Statement
C   WE SHALL FIRST SET UP COLUMN HEADINGS
          WRITE(6,66)
66        FORMAT('1',11X,'A',11X,'B',11X,'C',11X,'D',11X,'E',11X,'F',11X,'Y'//)
11        READ(5,12)A,B,C,D,E,F
12        FORMAT(6F3.1)
          SUM = A+B+C+D+E+F
          IF(SUM)40,30,40
40        X = SOLN(X,F,E,D,C,B,A)
          Y = 2.*X + SQRT(X)
          WRITE(6,21)(A,B,C,D,E,F,Y)
21        FORMAT(11X,6(F4.1,8X),E12.4)
          GO TO 11
30        STOP
          END
          REAL FUNCTION SOLN(XN,C5,C4,C3,C2,C1,C0)
          XN = 0.
          H = 1.0E-8
          H2 = H/2.
1         U = XN
          V = XN-H2
          P = C5+U*(C4+U*(C3+U*(C2+U*(C1+U*(C0)))))
          Q = C5+V*(C4+V*(C3+V*(C2+V*(C1+V*(C0)))))
          R = C5+XN*(C4+XN*(C3+XN*(C2+XN*(C1+XN*(C0)))))
          XN1 = XN-(R*H)/(P-Q)
          IF(ABS(XN1-XN)-.0001)60,61,60
61        XN = XN1
          GO TO 1
60        RETURN
          END
```

EXAMPLE 71

Write a FUNCTION subprogram which will compute the number of combinations of N things taken R at a time whenever a calling program executes the statement C = COMBO(N,R). Assume that N and R are integers and that N is greater than R.

Solution Procedure

The number of combinations of N things taken R at a time is given by the expression

$$C(n,r) = \frac{n!}{r!(n-r)!}$$

Straightforward computation of these factorials may produce overflows in large problems and will always involve unnecessary calculations. A more efficient procedure is to write the program so as to take advantage of the fact that the r low-order factors of $n!$ are equal to $r!$ and that the $n - r$ low-order factors of $n!$ are equal to $(n - r)!$. Thus, the computation is performed as follows:

$$\text{If } r \geq n - r,\ C(n,r) = \frac{(n)(n-1)\cdots(r+1)}{(n-r)!}$$

$$\text{If } r < n - r,\ C(n,r) = \frac{(n)(n-1)\cdots(n-r+1)}{r!}$$

Solution Program

Statement Number	*FORTRAN Statement*
	FUNCTION COMBO(I,J)
	COMBO = 1.
	IF (I−J) 1,1,2
1	K = I−J
	GO TO 3
2	K = J
3	DO 4 M = 1,K
4	COMBO = COMBO* (I−M+1.)/M
	RETURN
	END

Discussion

Here the independent nature of function subprograms is illustrated. COMBO is completely defined by this Solution Program and may be compiled. Unless referenced in a calling program by a statement of the form REAL = COMBO (I1,I2), however, COMBO is never executed. When referenced by such a statement, the value of COMBO is computed, and the RETURN statement sends control back to the statement immediately following the one that made such reference.

Note the use of mixed mode in Statement 4. Both integer and real numbers are intermixed in the evaluation of a real variable. First the quantity in parentheses is evaluated. Although I and M are integers, the result is real because the constant contains a decimal point. Next, the product COMBO * (I − M + 1.) is evaluated. Both of these quantities are real, and the result is real. Finally, the previous result is divided by M. The ratio of a real quantity and an integer quantity is real, according to the mixed-mode rule, so the result is real as required and has been computed properly. The thing to watch out for in mixed-mode calculations of this sort is a division resulting in an integer quotient. Normally, such a quotient, not having the ability to include fractional values, is not correct. A similar aspect of mixed-mode calculations that should not be overlooked is a double-precision calculation in which the full precision is not obtained because intermediate calculations produce integer or real numbers.

14.1.3 Subroutines

The *subroutine subprogram*, like the function subprogram, is written and may be compiled as an independent program: it is called by the main-line (calling) program as required; and it uses the RETURN statement to create the linkage back to the main line program. Except that SUBROUTINE is used instead of FUNCTION, its name and arguments are defined in the same way; and similarly to the FUNCTION subprogram, this type may contain any FORTRAN statement except a FUNCTION statement or another SUBROUTINE statement.

In contrast to the function subprogram, the subroutine subprogram, on the other hand, is not restricted to the return of a single quantity to the calling program. It may return two quantities, as in the following example, or any desired number, including none at all. In addition, the subroutine subprogram is called differently by the main line program. A special statement, CALL, is provided for this purpose, and the subprogram of this example would be called by a statement of the form

```
CALL DK (A,B,C,J)
```

The arguments in a CALL statement may be any type of constant, subscripted variable, or nonsubscripted variable, an array name, an arithmetic expression, or the name of a subprogram.

EXAMPLE 72

Write a subprogram that computes, given the coordinates of a point in two dimensions, the distance of the point from the origin and the location of the point relative to the coordinate system. Express the location of the point relative to the coordinate system according to the following code:

Code	*Location of Point*
0	Origin
1	First quadrant
2	Second quadrant
3	Third quadrant
4	Fourth quadrant
5	Positive X-axis
6	Positive Y-axis
7	Negative X-axis
8	Negative Y-axis

Solution Procedure

Both statement function subprograms and function subprograms are constrained to compute a single value. Two quantities are required in this example; therefore, a subroutine type of subprogram must be used.

The distance calculation is performed using the formula $D = \sqrt{X^2 + Y^2}$, the special case of the general formula $D = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$ that occurs when one of the points is at the origin. The determination of the location code is made as follows:

Conditions	*Code*
$X = 0,\ Y = 0$	0
$X > 0,\ Y > 0$	1
$X < 0,\ Y > 0$	2
$X < 0,\ Y < 0$	3
$X > 0,\ Y < 0$	4
$X > 0,\ Y = 0$	5
$X = 0,\ Y > 0$	6
$X < 0,\ Y = 0$	7
$X = 0,\ Y < 0$	8

Solution Program

Statement Number	*FORTRAN Statement*
	SUBROUTINE DK(X,Y,D,K)
	D = SQRT(X**2 + Y**2)
	K = 0
	IF (X) 1,2,3
1	IF (Y) 4,5,6
2	IF (Y) 7,8,9
3	IF (Y) 10,11,12
7	K = K + 1
5	K = K + 1
9	K = K + 1
11	K = K + 1
10	K = K + 1
4	K = K + 1
6	K = K + 1
12	K = K + 1
8	RETURN
	END

14.1.4 FORTRAN Library Subroutines

The previous example illustrates the facility provided by the FORTRAN system with respect to programmer-written subroutine subprograms. In addition to this facility, however, the FORTRAN system provides a number of built-in subroutine subprograms which, so to speak, are always on CALL. These are described below.

1. CALL SLITE (I)
   In many programs it is convenient to be able to set indicators of some sort as a means of *remembering* for future reference particular condi-

tions that were encountered. SLITE (for sense light) is a subprogram that provides this capacity. CALL SLITE (I), where I is an integer expression, turns "on" the Ith sense light if I = 1, 2, 3, or 4 and turns "off" all sense lights if I = 0.

2. CALL SLITET (I,J)
   This subprogram (sense light test) is used to test the various sense lights for the on or off condition so that alternative consequences can be pursued. Sense light I, where I is an integer expression equal to 1, 2, 3, or 4, is tested and turned off. If it were on, then J, an integer variable, is set to 1; if it were off, J is set to 2.
3. CALL OVERFL (J)
   The format of floating-point words and doublewords in System/360 is such that quantities greater than 16^{63} (approximately 10^{76}) cause a floating-point overflow condition. Similarly, quantities less than 16^{-64} (approximately 10^{-77}) cause a floating-point underflow condition. The OVERFL (for overflow) subprogram is used to test for an overflow or underflow condition where it is felt that, because of the nature of the problem, such might occur. J, an integer variable, is set to 1, 2, or 3 according to whether floating-point overflow has occurred, neither overflow nor underflow has occurred, or floating-point underflow has occurred, respectively.
4. CALL DVCHK (J)
   The attempt to divide by zero causes the divide check indicator to be turned on. The DVCHK (for divide check) subprogram is used to test this indicator. J, an integer variable, is set to 1 if the divide check indicator is on and to 2 if it is off. In either case the indicator is left in the off condition.
5. CALL EXIT
   This subprogram is used to terminate the execution of the object program. With respect to the particular program being executed, it is no different from STOP. With respect to the continued operation of the System/360, however, there may be a significant difference. STOP may cause a halt that requires operator intervention to overcome; EXIT sends control to the supervisory program so that immediate processing of the next job may begin.
6. CALL DUMP (A1,B1,F1, . . . ,AN,BN,FN)
   During FORTRAN compilations, memory assignments for variables and arrays are made as such are encountered by the processor. Dumps (essentially unformatted listings) of portions of memory may be specified with the DUMP subprogram in terms of these variable and array names. Each portion desired is specified by three arguments and the number of such portions is at the option of the programmer. The first two arguments, A and B, are the names that specify the upper and lower bounds, in either order, of a portion of memory to be dumped. The third argument, F, specifies the format according to the following code:

F	*Data Type*
0	Hexadecimal
4	Integer
5	Real
6	Double Precision

Omission of argument F produces hexadecimal output. This format, the most general, merely converts each group of four binary digits to the corresponding hexadecimal digit. Dumping is ordinarily done during the period in which a program is being tested. Not only the WRITE statement output is of interest at this state but also the values of key variables that indicate whether or not the program is performing as desired. The DUMP subprogram, which terminates execution of the object program in addition to producing output, is used to provide a record of the final status of these key variables.

7. CALL PDUMP (A1,B1,F1, . . . ,AN,BN,FN)
   This subprogram is identical to DUMP with one significant exception. PDUMP does *not* cause termination of object program execution. Consequently, it can be extremely useful in watching the behavior, for example, of an iterative program in which the numerical method used is supposed to bring about convergence of some variable to a limit. If calls to DUMP or PDUMP are included in a program, such calls are ordinarily removed after testing has been completed or, in the case of DUMP, are relegated to a part of the program which, if reached, necessitates both discontinuance of processing and a record for study of the conditions that led to that point.

14.1.5 Additional Subprogram Facilities

Normal entry into a subroutine subprogram is made by a CALL statement, and normal entry into a function subprogram is made by a function reference. In both cases, the execution of the subprogram begins with the first executable statement following the SUBROUTINE or FUNCTION statement. Execution may begin at other points, however, by using the ENTRY statement.

Normal exit from a subroutine or function subprogram is made by a RETURN statement which transfers control to the main-line program statement immediately following the one which transferred control to the subprogram. In the case of subroutine subprograms only, control may be transferred back to other points in the main-line program by using a special version of the RETURN statement.

EXAMPLE 73

Assume that in a main program the coordinates of two two-dimensional points have been developed. Call them (X1,Y1) and (X2,Y2). Write a subprogram that computes only the slope of the line segment joining the points when X2 $\geqslant$ X1 and

that computes both the slope and the length of the line segment when X2 < X1. Further, write the subprogram such that the return to the main-line program will be to any one of three distinct statements according to whether the computed slope is negative, zero, or positive.

Solution Program

```
Statement
Number    FORTRAN Statement

.         .
.         IF (X2 - X1) 1,2,2
1         CALL SUB1(X1,Y1,X2,Y2,D,S,10,20,30)
2         CALL SUB2(X1,Y1,X2,Y2,S,10,20,30)

.
10

.
20

.
30        .
          SUBROUTINE SUB1 (A1,B1,A2,B2,DIST,SLOPE,*,*,*)
          DIST = SQRT ((A2 -A1) ** 2 - (B2 - B1) * * 2)
          ENTRY SUB2 (A1,B1,A2,B2,SLOPE,*,*,*)
          SLOPE = (B2 - B1)/(A2 - A1)
.         IF (SLOPE) 1,2,3
1         RETURN 1
2         RETURN 2
3         RETURN 3
          END
```

Consider first the main-line program. The quantity X2 − X1 is tested and either SUB1 or SUB2 is called depending on the result. Included in the argument list of each CALL statement, in addition to the usual variable names, are three statement numbers. These identify the three points where control may be returned to the main program, depending on the value of the slope.

Now consider the subroutine subprogram. It appears to be named SUB1, and in its argument list are three asterisks that correspond to the statement numbers of the CALL statement. The third statement, however, is an ENTRY statement that defines the point of entry when SUB2 is called. It, too, contains three asterisks that correspond to the statement numbers in its CALL statement.

Whether both the length and the slope are computed, or only the slope, the

subprogram soon reaches one of the RETURN statements. The integer following each denotes which of the statement numbers in the argument list is the return address. If return to the main-line program is via RETURN 1, a branch to the first statement number in the list (that is, 10) is created; if via RETURN 2, a branch to statement 20 is created; and if via RETURN 3, a branch to statement 30 is created.

14.2 THE COMMON, EQUIVALENCE, AND EXTERNAL STATEMENTS

The ability within the FORTRAN language to compile subprograms as individual entities and to make the resultant object programs available to all the users of a particular installation can be of tremendous help to problem programmers. In the extreme case, a program might conceivably consist of specification statements, CALL statements, and the END statement. As a further aid in making such an operation feasible, even where the use of CALL statements is not so extreme, three additional specification statements are provided.

14.2.1 The COMMON Statement

The COMMON statement allows individual variables and arrays in calling programs and subprograms to share the same memory locations. Such sharing has two potential advantages. First, by conserving memory space, the "in-core" accommodation of larger programs is facilitated. Second, by using the same locations in both the calling program and one or more subprograms for data that are to be used by the subprogram or subprograms, the need for data movement is reduced.

In using the COMMON statement there are three prime rules that must be observed. First, it must be used in both or all programs which are to share memory locations. Second, the order in which the variables and arrays are listed in the statement determines the order in which memory allocations are made. Finally, memory is shared not on a variable-by-variable basis but on the basis of the space required by each variable. Therefore, a COMMON statement listing two real variables and a COMMON statement listing one double-precision variable would cause both real variables to be assigned the same memory space as the single double-precision variable. It is suggested that in listing variables in a COMMON statement the double-precision variables be first, the real variables be second, and the integer variables be third.

Some examples of usage follow:

Calling Program	*Subprogram*
DOUBLE PRECISION ZBAR	DOUBLE PRECISION A,B,C
COMMON ZBAR, X,Y,Z,I	COMMON A,B,C

Three doublewords of memory are shared. The calling program places ZBAR in the first, X and Y in the second, and Z and I in the third. The subprogram places A, B, and C in the first, second, and third, respectively.

```
Calling Program          Subprogram
COMMON A(10), B(10,10)   COMMON Z(10), Y(50), X(50)
```

The A and Z arrays share the same 10 words. The Y and X arrays share the 100 words required by the B array. Because two-dimensional arrays are stored row by row, the first five rows of B are shared by Y, and the last five are shared by X.

```
Calling Program   Subprogram
COMMON A,B,C      COMMON U,V,W,X,Y,Z
COMMON D,E
COMMON F
```

The effect of more than one COMMON statement in the same program is the same as if all the arguments were in the first and in the same order. Here memory is shared by A and U, B and V, C and W, D and X, E and Y, and F and Z.

14.2.2 The EQUIVALENCE Statement

The EQUIVALENCE statement allows individual variables and arrays within the same program to share the same memory locations. It frequently happens that variable and array names used early in a program are not used again later when space is required for other data or results. While the same names could be used over again for the new requirements, this is not always desirable. It may complicate the indexing of subscripts, and it discards the advantage of giving variables meaningful names. The sample statement

```
EQUIVALENCE (A,B), (C,D(1)), (E(2,4), F(3,1,2), G)
```

assigns equivalent memory locations to A and B, to C and D(1), and to E(2,4), F(3,1,2), and G.

Variables or arrays not mentioned in EQUIVALENCE statements are assigned unique locations, but arrays, of course, are arranged in a regular manner. EQUIVALENCE statements, if such are made, must take this arrangement into consideration.

14.2.3 The EXTERNAL Statement

The usefulness of subprograms is enhanced by the ability to use the name of one in the argument list of another. Whenever this is done, the EXTERNAL statement must be used in the main program. The statement

```
EXTERNAL DIST, INVERT
```

for example, specifies that both DIST and INVERT are the names of subprograms that are used in the argument lists of other subprograms. Thus a calling-program statement such as CALL MATRIX (M,N,INVERT), where INVERT is a subprogram name, requires the use of EXTERNAL INVERT in the same program. A calling-program statement such as CALL TIME (50,S,DIST(T)), where DIST is a subprogram name, however, does not require the use of EXTERNAL. The reason for this is that DIST is not being used as an argument: DIST(T) is evaluated first, and the result is what is used as the argument.

14.3 SOURCE-PROGRAM STATEMENT ORDER

As the FORTRAN compiler translates each statement of the source program, it must have certain information available to it. Some of this information is included in the statement being translated; other information, however, must have been supplied in prior statements. For this reason, a summary of statement-order rules is presented here.

While the distinction is not quite as clear-cut as this, each FORTRAN program or subprogram might be considered as a group of specification statements followed by a group of executable statements followed by the END statement. Specification statements include: INTEGER, REAL, DOUBLE PRECISION, DIMENSION, FORMAT, COMMON, EQUIVALENCE, EXTERNAL, FUNCTION, SUBROUTINE, and the arithmetic statement that defines a statement function. All of these may be used in both calling programs and subprograms except EXTERNAL (calling programs only), FUNCTION (function subprograms only), and SUBROUTINE (subroutine subprograms only). All of these must be used before the first executable statement of each program in which they are used except FORMAT, which may be placed anywhere among the executable statements.

With respect to the statements that must be used before the first executable statement of a program, there are three rules to follow. First, statements which describe data must come before statements which refer to those data; second, statements which define statement functions must be placed after other specification statements; and third, whenever programs utilize arrays, the dimensions of such arrays should be declared in the first specification statement in which they appear. Declarations of array dimensions may be accomplished using the INTEGER, REAL, DOUBLE PRECISION, DIMENSION, and/or COMMON statements.

When a FUNCTION subprogram is specified, the FUNCTION statement must be first; otherwise, the statement-order rules are the same as for calling programs. Similarly, when a SUBROUTINE subprogram is specified, the SUBROUTINE statement must be first; otherwise, the calling-program statement-order rules apply. Except as the first statement, FUNCTION and SUBROUTINE

subprograms cannot contain the FUNCTION and SUBROUTINE statements.

Executable statements include: Arithmetic, Unconditional GO TO, Computed GO TO, Arithmetic IF, DO, READ, WRITE, END FILE, REWIND, BACKSPACE, CONTINUE, PAUSE, STOP, RETURN, CALL, and the arithmetic statement that calls statement function and FUNCTION subprograms. All of these may be used in both calling programs and subprograms except STOP (calling program only) and RETURN (function and subroutine subprograms only). Both STOP and RETURN may appear any number of times in the programs in which they appear.

14.4 COMPLEX NUMBERS

Numerous application areas, such as those associated with the treatment of alternating currents and voltages, for example, require the use of complex numbers. In previous versions of FORTRAN, complex numbers could only be handled indirectly via the FORTRAN language; that is, the real and imaginary parts could be programmed separately. For example, the formation of the following product

$$x + yi = (a + bi)\ (c + di)$$

could be programmed in two steps. Thus

```
X = A*C - B*D
Y = B*C + A*D
```

A more natural procedure, however, would be to program this computation as the single operation it is. To do so requires the ability to define complex constants and variables. Complex constants are expressed as pairs of real or double-precision constants enclosed in parentheses and separated by commas. Complex variables are designated by the COMPLEX specification statement in a manner parallel to the use of INTEGER, REAL, and DOUBLE PRECISION. Thus,

```
COMPLEX A,K,D
```

specifies that A, K, and D are complex variables.

COMPLEX specification statements have two additional capabilities. They may be used to specify the number of bytes occupied by a variable, and they may be used to specify the initial value of a variable. For example, the statement

```
COMPLEX U/(4.,8.)/,V/(2.,-3)/,X
```

defines U, V, and X as complex variables and specifies the initial values of U and V as $4 + 8i$ and $2 - 3i$, respectively. Because no statement to the contrary is made, both the real and imaginary parts of each variable are made four bytes in length. If it were desired to have each eight bytes in length, the following form could have been used:

```
COMPLEX*16U/(4.,8.)/,V/(2.,-3)/,X
```

EXAMPLE 74

If U = 4 + 8*i* and V = 2 − 3*i*, compute X = $(3U^3 + 4U^2 - 8U)/(6V^4 - 3V^2 + 10)$.

Solution Program

Statement Number *FORTRAN Statement*

```
      COMPLEX U/(4.,8.)/,V/(2.,-3.)/,X
      X = (3.*U**3 + 4.*U**2 - 8.*U)/(6.*V**4 - 3*V**2 + 10.)
      STOP
      END
```

Each of the variable types has an implied, or standard, length and an optional length that may be chosen via such a specification statement. The standard length of integers is four bytes; the optional length is two. The standard length of real numbers is four bytes; the optional length is eight (making the use of DOUBLE PRECISION unnecessary). As already stated, the standard length of complex numbers is eight bytes; the optional length is 16. Logical variables, discussed in Sec. 14.5, have a standard length of four bytes and an optional length of one. The following statements indicate how the various length specifications may be mixed.

```
COMPLEX A,B,C*16
REAL*8 X,Y*4,Z
INTEGER D*2,P,Q,R*2
LOGICAL*1 T,F                          (See 14.5)
```

These four statements specify that A and B are complex variables eight bytes in length and that C is a complex variable 16 bytes in length; that X and Z are eight-byte real variables while Y is a real variable of standard length; that D and R are integer variables each requiring two bytes of storage while P and Q occupy four bytes each; the logical variables T and F occupy a single byte each.

If explicit specification statements are not used by the programmer to denote the variable types, the type implied by the first letter of each variable name is used instead. The normal implication is that variable names that begin with I, J, K, L, M, or N denote integer variables and variable names that begin with all other letters denote real variables. This implication is a carryover, for purposes of compatibility, from earlier dialects of the FORTRAN language in which only integer and real variables were present. An advanced feature, the IMPLICIT statement, allows the programmer to set his own *normal* implications. For example,

```
IMPLICIT INTEGER (A-H), REAL (I-K), COMPLEX*16 (L-Z)
```

sets the normal implication to denote as four-byte integers all symbols beginning with A–H; as four-byte reals, all symbols beginning with I–K; and as 16-byte complex variables, all symbols beginning with L–Z.

TABLE 14.1 BUILT-IN FUNCTIONS FOR COMPLEX VARIABLES

FORTRAN Usage	Mathematical Equivalent
1. C = CEXP(X)	$C = e^X = e^{X_1}\cos X_2 + e^{X_1}\sin X_2 i$
CD = CDEXP(Y)	$CD = e^Y = e^{Y_1}\cos Y_2 + e^{Y_1}\sin Y_2 i$
2. C = CLOG(X)	$C = \log_e X = \log_e X_1 + \tan^{-1}(X_2/X_1)i$
CD = CDLOG(Y)	$CD = \log_e Y = \log_e Y_1 + \tan^{-1}(Y_2/Y_1)i$
3. C = CLOG10(X)	$C = \log_{10} X = \log_{10} X + \tan^{-1}(X_2/X_1)i$
CD = CDLG10(Y)	$CD = \log_{10} Y = \log_{10} Y_1 + \tan^{-1}(Y_2/Y_1)i$
4. C = CSIN(X)	$C = \sin X = \sin X_1 \cosh X_2 + \cos X_1 \sinh X_2 i$
CD = CDSIN(Y)	$CD = \sin Y = \sin Y_1 \cosh Y_2 + \cos Y_1 \sinh Y_2 i$
5. C = CCOS(X)	$C = \cos X = \cos X_1 \cosh X_2 - \sin X_1 \sinh X_2 i$
CD = CDCOS(Y)	$CD = \cos Y = \cos Y_1 \cosh Y_2 - \sin X_1 \sinh X_2 i$
6. C = CSQRT(X)	$C = \sqrt{X} = (X_1^2 + X_2^2)^{1/4}(\cos 1/2\tan^{-1}(X_2/X_1) + \sin 1\ 2\tan^{-1}(X_2/X_1)i)$
CD = CDSQRT(Y)	$CD = \sqrt{Y} = (Y_1^2 + Y_2^2)^{1/4}(\cos 1/2\tan^{-1}(Y_2/Y_1) + \sin 1\ 2\tan^{-1}(Y_2/Y_1)i)$
7. D = CABS(X)	$D = (X_1^2 + X_2^2)^{1/2}$
8. A = REAL(X)	$A = X_1$
A = AIMAG(X)	$A = X_2$
9. C = CMPLX(B1,B2)	$C = B1 + B2i$
CD = DCMPLX (E1,E2)	$CD = E1 + E2i$
10. C = CONJG(X)	$C = X_1 - X_2 i$
CD = DCONJG(Y)	$CD = Y_1 - Y_2 i$

The use of complex constants and variables in subprograms is no different from that of other types of numbers. Added to the built-in mathematical function subprograms, however, are additional such subprograms specifically designed for complex variables. These are described in Table 14.1. In these descriptions the symbols C, CD, D, K, and A represent variables of the following respective types: complex (length 8), complex (length 16), real (length 8), integer (length 2), and real (length 4). The symbols B, E, X, and Y denote expressions, respectively, of the following types: real (length 4), real (length 8), complex (length 8), and complex (length 16). X_1 and X_2 represent the real and imaginary components of X; Y_1 and Y_2 represent the real and imaginary components of Y.

14.5 LOGICAL QUANTITIES

Constants and variables encountered thus far have been numeric, as have been the expressions compounded of these elements. In addition, the operations of algebra, when applied to these numeric factors, have produced numeric results. Logical constants and variables, on the other hand, are nonnumeric in nature and, may have only two possible values: .TRUE. and .FALSE. (the periods allow the FORTRAN compiler to distinguish these values from variable names). Logical expressions are those whose value is either .TRUE. or .FALSE. and which may be formed in three ways: (1) by a combination of arithmetic expressions (integer or real only) with *relational*

operators, (2) by a combination of logical constants and variables with *logical operators*, or (3) by a combination of the above two forms.

There are six relational operators, as shown:

Relational Operator	*Definition*
.GT.	Greater than
.GE.	Greater than or equal to
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to

If I and J are integer variables, for example, the logical expression I.LT.J + 2 is .TRUE. if the numeric value of I is less than the numeric value 2 greater than J and is .FALSE. otherwise.

There are three logical operators: .NOT., .AND., and .OR.. Each is defined in the following table in conjunction with all possible values of two logical variables P and Q.

If both P and Q are .TRUE.

.NOT. P is .FALSE.
.NOT. Q is .FALSE.
P .AND. Q is .TRUE.
P .OR. Q is .TRUE.

If P is .TRUE. and Q is .FALSE.

.NOT. P is .FALSE.
.NOT. Q is .TRUE.
P .AND. Q is .FALSE.
P .OR. Q is .TRUE.

If P is .FALSE. and Q is .TRUE.

.NOT. P is .TRUE.
.NOT. Q is .FALSE.
P .AND. Q is .FALSE.
P .OR. Q is .TRUE.

If both P and Q are .FALSE.

.NOT. P is .TRUE.
.NOT. Q is .TRUE.
P .AND. Q is .FALSE.
P .OR. Q is .FALSE.

The LOGICAL specification statement is used in the same way as INTEGER, REAL, and COMPLEX to denote explicitly the usage of variable, array, or function names, to declare the length associated with each such name, to set such quantities to their initial values, and to define, where appropriate, the numbers of dimensions and the sizes of arrays. The implied length of

logical variables is 4; the optional length is 1. The following statement, for example, declares A and B to be logical variables:

```
LOGICAL A(5)/2* .TRUE., 3* .FALSE./, B*1
```

A, whose length is 4, is a single-dimension array of five elements, of which the first two are initially .TRUE. and the remainder initially .FALSE.. Logical variable B has a length of 1.

EXAMPLE 75

A deck of 100 cards is punched with a T or F in column 1, with a 2-digit integer in columns 2 and 3, with a 3-digit integer in columns 4–6, and with other data in the remainder. Each card contains data about a person, with column 1 denoting sex (T = male, F = female), columns 2 and 3 denoting age to the nearest year, and columns 4–6 denoting weight to the nearest pound.

Read the cards and punch out a new deck in which columns 7–80 are the same, punch the word 'PASSED' in columns 1–6 of those cards which represent a male 40 years of age or over and weighing 160 pounds or less or a female 36 years of age or over and weighing 120 pounds or less. Otherwise, punch the word 'FAILED' in columns 1–6.

Solution Procedure

Without the availability of logical variables, the processing required to determine whether or not each person meets the appropriate criteria would involve a set of IF statements. By using a logical expression, the test is applied in a single statement.

Solution Program

```
Statement
Number   FORTRAN Statement
         LOGICAL*1 TEST,SEX
         INTEGER*2 AGE, WEIGHT, ALPHA(37)
         DO 1 I = 1,100
         READ (5,10) SEX,AGE,WEIGHT,ALPHA
10       FORMAT (L1, I2,I3,37A2)
         TEST = (SEX.AND.(AGE.GE.40).AND.(WEIGHT.LE.160))
        1.OR.(.NOT.SEX.AND.(AGE.GE.36).AND.(WEIGHT.LE.120))
         IF(TEST) GO TO 2
         WRITE(7,20)ALPHA
         GO TO 1
2        WRITE(7,30)ALPHA
1        CONTINUE
         STOP
20       FORMAT ('FAILED',37A2)
30       FORMAT ('PASSED',47A2)
         END
```

The logical capabilities of FORTRAN are illustrated in several ways in the Solution Program. The LOGICAL specification statement is used to define TEST and

SEX as logical variables; the L format code is used in reading the logical variable SEX from each card; the logical replacement statement TEST = ... is used to compute whether or not the criteria have been met; and the logical IF statement is used to take alternative courses of action based on the value of TEST.

The logical variable SEX, along with the integer variables AGE and WEIGHT and the alphameric array ALPHA, is read from a card. Logical variables treated in this manner must be punched as T or F and are read using the L format code. Thus the variable SEX is set to .TRUE. or .FALSE. accordingly as column 1 of the card is punched T or F. The L format code, however, may be used in a more general way. For example, the following statement states that logical variables, each located in a three-character field, are to be read or written:

```
FORMAT (10L3)
```

On input, the first T or F encountered in a field sets the value of the associated variable. On output, leading blanks are inserted in fields greater than one character.

The replacement statement TEST = ... serves to set the value of the logical variable TEST to .TRUE. or .FALSE. according to the value of the logical expression on the right. In structure, then, it is equivalent to arithmetic replacement statements in which the value of the arithmetic expression on the right-hand side of the equals sign replaces the value of the arithmetic variable on the left-hand side. While its length is such that the statement must be continued on the following line, the use of relational and logical operators produces a considerably shorter program.

Parentheses have been placed in such a way as to make interpretation of this statement easier. TEST is .TRUE. if either the quantity on the remainder of the first line or the quantity after .OR. on the second line is .TRUE.. Each of these compound quantities, in turn, is .TRUE. only if all three of its components are .TRUE.. In each case, the first of these three components is determined by the value of SEX; the second, by the relationship of AGE to a given criterion; and the third, by the relationship of WEIGHT to a given criterion. A data card for a 150-pound man 25 years of age, for example, would produce the following result (for brevity, T and F are used as abbreviations for .TRUE. and .FALSE.):

```
F = (T .AND. F .AND. T) .OR. (F .AND. F .AND. F)
```

On the other hand, a data card for a 110-pound woman 40 years of age would produce the following result:

```
T = (F .AND. T .AND. T) .OR. (T .AND. T .AND. T)
```

Where parentheses are not used to establish clearly the order of computation, the order is as follows: arithmetic computations (in the order previously discussed), relational operators (.LT., .LE., .EQ., .NE., .GT., .GE.), .NOT., .AND., and .OR..

The particular WRITE statement in the Solution Program that is executed depends on the IF statement. The IF statement used here is the Logical IF rather than the Arithmetic IF. It is used in the following way. A logical variable or expression is placed within the parentheses and another FORTRAN statement is placed to the right. If the logical variable or expression is .TRUE., the statement to the right is executed next; if it is .FALSE., the following statement is executed next. The statement placed to the right may be any FORTRAN statement except a specification statement, DO statement, or another Logical IF statement. In the Solution Program, there

was no real need for the replacement statement TEST = ... because the expression to the right could have been placed within the parentheses of the Logical IF statement. For clarity, however, this was not done.

14.6 THE G FORMAT SPECIFICATION

Suppose variables A, B, and C have been specified as integer, real, and logical variables, respectively, and values for each have been obtained. Consider the following pair of statements:

```
      WRITE (2,10) A, B, C
10    FORMAT (I8, F15.3, L1)
```

Notice that the use of I, F, and L is largely redundant; the types of the variables A, B, and C have been previously specified and contain the same information. It is possible, therefore, to devise a generalized format code with the capability of being applicable to integer, real, complex, and logical data. The G format code is provided for this purpose.

EXAMPLE 76

Purely as an exercise in the use of the G format code, write a program to print out the values of the following set of variables:

Variable name	*Variable Type*	*Specific Value*
A	Logical	.TRUE.
B	Logical	.FALSE.
C	Integer	423
D	Real	349.209
E	Real	.0000006429
F	Complex	$32.54 - 84.13i$

Solution Program

Statement Number	*FORTRAN Statement*
	LOGICAL A/.TRUE./, B/.FALSE./
	INTEGER C/423/
	REAL D/349.209/, E/6.429E−7/
	COMPLEX F/(32.54,−84.13)/
	WRITE (6,1) A,B,C,D,E,F
1	FORMAT('1',3G4.2G12.6,2G10.4)
	STOP
	END

The resultant print line, in which ␣ represents a blank, is produced by the Solution Program:

␣␣␣T␣␣␣F␣423␣349.209␣␣␣␣␣.642900E−06␣32.540000−84.130000

If the form of the G format code is construed as *aGb.c*, where not every element need always be present, *a* is the previously encountered multiplier that states how many times the remainder of the form is to be applied; *b* is the previously encountered total field size; and *c* is the number of significant digits to be transmitted, rather than the number of digits to the right of the decimal point as in the D, E, and F format codes. Both *c* and the preceding decimal point are not used for logical or integer data and, if present, are ignored.

In the solution program, 3G4 denotes three four-position fields of either logical or integer data or both, and the WRITE statement list, in combination with the explicit specification statements, shows that the first two data are logical and the third datum is an integer. The output line shows all three data right-justified in their respective fields.

Next, 2G12.6 denotes two 12-position fields of either real or complex data, each field containing six significant digits. These fields, D and E, are both real. D(349.209) fits without alteration into the allocated space. Note that it is placed in such a way that four positions to the right are reserved in the event that an exponent must be used. E (.0000006429) must use the exponent form to fit into the allocated space.

Finally, 2G10.4 denotes two 10-position fields of either real or complex data, each field containing four significant digits. The WRITE statement list indicates the variable to be transmitted is F, a complex variable. In this case, both the real and imaginary parts of this datum are transmitted as if they were two independent real data.

In addition to the above comments which apply to the Solution Program, two last comments about the G format code must be made. First, it is always required to include the decimal point in real data being read with the G format code, in contrast to the D, E, and F format codes where it is optional. Second, if a datum includes the D decimal exponent, it is transferred with this D decimal exponent.

14.7 OTHER FEATURES

In addition to the above types of advanced features, there are a few other miscellaneous ones. They are not as generally useful, but they may be quite advantageous in certain circumstances.

1. Ordinarily it is not allowed to branch into the range of a DO; a DO loop must be begun at the beginning. It is permitted to branch out of the innermost of a next of DO loops and then return to that innermost loop, however, as long as none of the DO indices are altered.
2. The dimensionality of arrays is increased from three to seven.
3. Another form of the PAUSE statement is available. It is PAUSE 'message' where the message between the single quotation marks is displayed to the operator when this statement is reached. Otherwise, this form is similar to the forms PAUSE and PAUSE *n*.
4. The parameters in the parentheses following the READ (*a,b*) *list* statement may be augmented by two, resulting in the following variation: READ (*a,b*,END = *c*,ERR = *d*) *list*. The interpretation of this statement is: read the list of variables from input device *a* according to FORMAT

statement *b*, but transfer to *d* if an error is recognized; if no errors are encountered, transfer to *c* upon sensing the end of the data set.

5. A new READ statement is included. It is READ (*a*,*x*) where *a* is a reference to an input device as before but where *x* is a variable name, not the number of a FORMAT statement. This variable name *x* appears in a new statement, the NAMELIST statement, and a set of variable or array names appears with it. When this statement is used in place of a FORMAT specification, the data being read must be organized in a special way. An example is in order.

Statement Number	*FORTRAN Statement*
	.
	NAMELIST/DATA/A,B,I/TEST/K
	.
	READ (1,DATA)
	.
	READ (1,TEST)

Note that the first READ specifies that values for A, B, and I are to be read and that the second READ specifies that a value for K is to be read but that nowhere is the format of these variables specified. The format is specified with the input data themselves, as illustrated by the two sets of data cards below (using this type of data, column 1 of each data card is ignored; the cards shown are punched starting in column 2).

First Set

First Card	&DATA
Second Card	A = 42.13, B = 30.28
Third Card	I = 10
Fourth Card	&END

Second Set

First Card	&TEST
Second Card	K = 36
Third Card	&END

6. A variation of the Computed GO TO statement is the Assigned GO TO statement. An example is:

```
GO TO N, (10,3,42,65)
```

When N = 10, control is transferred to statement 10; when N = 3, control is transferred to statement 3, and so forth. The value of N may be set by the special ASSIGN statement only. For example,

```
ASSIGN 10 TO N
```

sets N = 10 until a subsequent assignment is made.

7. In addition to the usage of COMMON previously described (known as *blank* COMMON), it is possible to name common areas in which specific variables are stored. Thus

```
COMMON/ARG/PI,DELTA,X,Z
```

specifies that the variables PI, DELTA, X, and Z are to be stored in the common area labeled ARG. Such areas are called *labeled* COMMON.

8. The BLOCK DATA subprogram may be used to enter data into a named (labeled) common area. This subprogram starts with the BLOCK DATA statement, follows this with the type of labeled COMMON statement shown above, and ends with END. In between, no executable statements are allowed, only specification statements such as REAL, DIMENSION, COMPLEX. For example,

Statement Number	*FORTRAN Statement*
	BLOCK DATA
	COMMON/ARG/PI,DELTA,X,Z
	REAL PI/3.14159/,DELTA/.001/
	COMPLEX X/(3.42,−8.39)/
	REAL*8 Z(4)/4.3,8.9,6.2,9.0/
	END

This subprogram stores the given values into the common area ARG according to the various specification statements. It is not necessary to specify a value for every variable in the labeled common area.

PROBLEMS

1. In what three ways may subprogram capabilities be of use to the problem programmer?
2. Discuss the characteristics of the two classes and five types of subprograms in the FORTRAN language.
3. What are the statements included in the FORTRAN language to facilitate the use of subprograms, and what is the purpose of each?

4. Versions of the FORTRAN language for large System/360 installations include advanced features. What are the advantages and disadvantages of having advanced features available?
5. What are the relative advantages and disadvantages of the Arithmetic IF statement with respect to the Logical IF statement?
6. Write a program to compute I = MOD(J1,J2) without using this or any other built-in subprogram.
7. Write a program that uses built-in subprograms to evaluate each of the following functions from X = 0 to X = 5 at steps of 0.1 and from Y = 0 to Y = 1 at steps of 0.05.
   (a) $z = e^x \tan y$
   (b) $z = (x + |\sin y|)^{1/4}$
   (c) $z = e^{\log_e(1 + x + y)}$
8. Write a program to integrate the function $y = x^2 e^x \sin x$ from $x = 0$ to $x = 1$ by using the Trapezoidal Rule. That is, divide the range of x into n parts, each of length h. Then the area can be expressed as

$$A = \frac{h(y_0 + y_n)}{2} h(y_1 + y_2 + y_3 + \cdots + y_{n-1})$$

   Define y as a statement function and use it whenever one of the y_i must be evaluated.
9. Write a program to evaluate the function $y = (P - Q)!\ P!\ Q!$ for all integral values of P and Q from 1 through 10 and for all cases where $P \geq Q$. Define, write, and use a function subprogram for the computation of the factorials.
10. Write a subroutine subprogram to compute the unknown side and angles of a triangle, given two sides and the included angle. [*Hints:* Use the Law of Cosines to find the unknown side, then the Law of Sines to find the unknown angles.]
11. Write a subroutine subprogram to compute the area, perimeter, and angles of a triangle, given the three sides. [*Hint:* If the three sides are a, b, and c, the area may be computed as

$$A = [S(S - a)(S - b)(S - c)]^{1/2}$$

    where $S = (a + b + c)/2$.]
12. Write a subroutine subprogram to compute the last term and the sum of both arithmetic and geometric series, given the first term, difference or ration, number of terms, and a code to designate the type of series. If desired, write this as a multiple-entry subprogram.
13. Write a subroutine subprogram to compute the coefficients of $(1 + x + x^2 + x^3)^3$. Then write a more general subroutine subprogram to compute the coefficients of $(1 + x + x^2 + x^3 + \cdots + x^p)^n$ for values of p and n such that $np \leq 500$.
14. A ray of light passing from one medium to another is refracted according to the formula:

$$n_i \sin \Phi_i = n_r \sin \Phi_r$$

where n_i and n_r are the indices of refraction of the two media and where Φ_i and Φ_r are the angles between the ray and the normal to the boundary at the point of intersection for the incident and refracted rays, respectively. Write a subroutine subprogram to compute Φ_r, given n_i, n_r, and Φ_i. No refraction can occur if $(n_i/n_r) \sin \Phi_i > 1$; the subprogram should treat this occurrence as a special case.

15. Write a subroutine subprogram to multiply two rectangular matrices together. The maximum size of each is 30 rows and 30 columns.

FURTHER READING

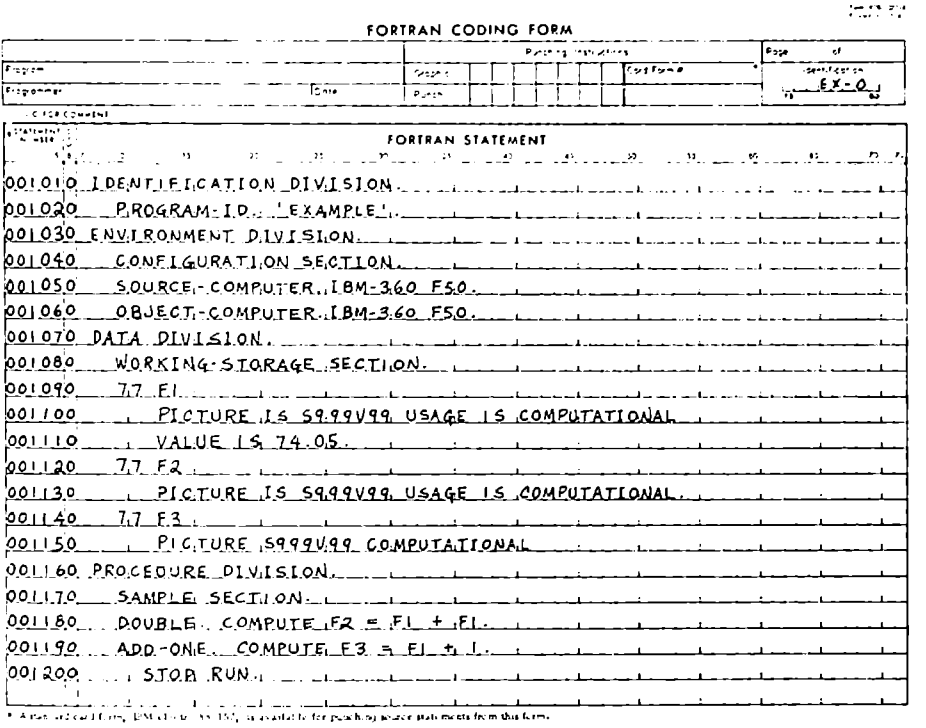
1. *Tape/Disk Operating Systems—Basic FORTRAN IV Language*, Form C28-6629. IBM Corp., White Plains, N.Y.
2. *Tape/Disk Operating Systems—FORTRAN IV Programmer's Guide*, Form C28-5038. IBM Corp., White Plains, N.Y.
3. *Operating System—FORTRAN IV Language*, Form C28-6515. IBM Corp., White Plains, N.Y.
4. *Operating System—FORTRAN IV Programmer's Guide*, Forms C28-6603, C28-6639, C28-6602. IBM Corp., White Plains, N.Y.

Chapter 15

The COBOL Programming System

COBOL (COmmon Business Oriented Language) provides strong capabilities for handling applications whose chief function is the maintenance of large and frequently complex data files and the production of management reports and accounting and other documents based on such files. In order for COBOL to be able to provide its data-processing and report writing capabilities, it contains somewhat more statements, rules, and formats than does FORTRAN. In practice this does not unduly burden the problem programmer.

The uses for which COBOL was designed have led to a system whose source programs give the appearance of English, in the same way that FORTRAN source programs give the appearance of algebra (or Assembler Language source programs give the appearance of an alphameric cryptogram). It should be noted that, while the extensive use of English words and constructions in COBOL is a significant aid to its understanding, use, and retention, it contains the same sort of strict requirements on the way such words and constructions are used by the problem programmer as are imposed by the previously encountered programming systems.



FORTRAN CODING FORM

| | | |

Numeric Digits	*Special Characters*			*Letters of the Alphabet*
0-9	+	<	(	A-Z
	−	>	)	
	*	.	:	
	/		$	
	=		b	

This does not mean that other machine-readable characters cannot be handled by COBOL programs. Instead, it defines the repertoire of acceptable characters for the program itself.

15.1.2 Coding Format

The COBOL coding sheet is designed in accordance with a prescribed card format which consists of four parts: (1) sequence number (columns 1-6), (2) continuation indicator (column 7), (3) source-program statements (columns 8-72), and (4) program-identification code (columns 73-80). Of these, the first and last are optional and need never be used, while the second is rarely used.

Note that the sequence number on the program sheet is subdivided into a page number (columns 1-3) and a serial number (columns 4-6) but that only the latter appears with each line. This is to allow the problem programmer to use the first half of the sequence number as the identification for each page of his program without forcing him to write it down unnecessarily. If sequence numbers are used, only numeric digits are allowed. The compiler will recognize these numbers and check that the source program cards are in their intended sequence. If columns 1-6 are left blank, no sequence check can or will be made. On the COBOL Program Sheet, note also that the optional program identification code, if used, need only be written once per page.

The COBOL language has been designed to allow the continuation of statements from one line (card) to another. Under all circumstances except one, however, this is accomplished without the use of the continuation indicator. In the case of nonnumeric literals (described later), the need to use this continuation indicator may occur. When this happens, a hyphen in column 7 denotes the usage.

The problem programmers's source program statements are entered into columns 8-72. This region, which is the only part of the coding form shown in subsequent examples, is subdivided into two *margins*, Margin A and Margin B. *Margin A* consists of columns 8-11, and *Margin B* consists of columns 12-72. The significance of these margins for the problem programmer is that certain elements of COBOL must begin, but need not be

contained, within Margin A, while other elements must begin, and must be contained, within Margin B.

Columns 73–80 of the COBOL program sheet have no effect on either the compiler or object program but may be used by the problem programmer for identification purposes.

Two types of explanatory remarks and comments may be included. These are for the user's benefit and are not processed by COBOL. A section of narrative material may be included in the IDENTIFICATION DIVISION. It begins with REMARKS and a period, followed by the desired comments and is terminated by the appearance of an entry in Margin A.

The second type of comment may appear in the PROCEDURE DIVISION of the program and begins with the keyword NOTE.

15.1.3 Program Organization

Every COBOL program consists of four *divisions:* IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION (see Figure 15.1). Not only must each be present in every COBOL program, but their order must be as indicated above; the division header statements must be explicitly included as shown; and such division header statements must begin in Margin A. A division header is always terminated by a period followed by blanks through column 72.

IDENTIFICATION DIVISION always includes PROGRAM-ID (which must begin in Margin A), followed by a period, space, and then the program name. (In general, always leave a space after a period.) Program name must begin with a letter of the alphabet, must contain from one through eight letters and/or numeric digits, and must be enclosed in single quotation marks or apostrophes. As will be illustrated later, the IDENTIFICATION DIVISION may optionally include other keywords and associated program information. These keywords, which must begin in Margin A, are: AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, SECURITY, and REMARKS. Each of these keywords, if used, is terminated by a period and a space, and then followed on either the same or following lines by additional identification information.

ENVIRONMENT DIVISION always includes CONFIGURATION SECTION, as shown in Figure 15.1, and it also includes INPUT-OUTPUT SECTION when I/O devices and files are utilized. In the general case, then, this division is composed of two *sections.* CONFIGURATION SECTION always contains the *paragraphs* that specify SOURCE-COMPUTER and OBJECT-COMPUTER, and may include additional information; INPUT-OUTPUT SECTION, not yet illustrated by example but, if used, contains two paragraphs: FILE-CONTROL and I-O-CONTROL. The latter will be described when illustrated. Note now, however, that all throughout the COBOL language the names of divisions, sections,

and paragraphs must begin in Margin A. Whereas the names of divisions and sections must appear on a line by themselves, the name of a paragraph may be followed on the same line, after a period and a space, by additional information which must not start before Margin B.

DATA DIVISION provides the problem programmer with the means to define the content and organization of data files, to define constants and allocate memory, to create linkage information for subprograms, and to specify other items as well. Only the WORKING-STORAGE SECTION, the rough equivalent of the DS statements of Assembler Language, is illustrated in Figure 15.1 and that, only in small part. In this illustration, F1, F2, and F3 are defined as independent working storage fields (level number 77) in fixed-point binary format (COMPUTATIONAL) whose decimal equivalent, with format given by a PICTURE, contains a sign and five digits with two digits understood to be to the right of the decimal point (s999v99). The initial value of F1 is specified as 74.05, while the values of F2 and F3 are not given. Variations in the way the statements may be written are indicated.

PROCEDURE DIVISION contains subdivisions (sections, paragraphs, etc.) which specify the processing to be carried out. Unlike the subdivisions of the other three divisions, sections, and paragraphs of this division are named by the problem programmer according to the requirements of particular programs. Such names (SAMPLE-SECTION, DOUBLE, and ADD-ONE in the example in Figure 15.1) serve not only as a means of documenting the various procedures within the PROCEDURE DIVISION, but they also serve the same function as statement numbers in FORTRAN: that is, branching statements include such names directly. Programmer-supplied names of this sort may be from 1 through 31 characters in length. Each such name is contructed from the character set consisting of the numeric digits, the alphabet, and the hyphen (-), except that the first or last character may not be a hyphen.

Within the sections and paragraphs of the PROCEDURE DIVISION, *statements* and *sentences* specify specific procedures. These statements and sentences utilize a set of structured forms and include words reserved by the COBOL language. Among these are COBOL *verbs*, such as COMPUTE and STOP in the example, which form the basis for input/output, data manipulation, computation, and branching actions. STOP RUN is an optional form of usage of the STOP verb; it specifies that execution of the object program is to be terminated and that control of the computer is to be returned to the operating system.

15.1.4 Literals

In addition to the program constants that are identified by name in the DATA DIVISION of a COBOL program (F1 in the example), constants may also be defined by their own identities (such as 1 in the statement

COMPUTE F3 = F1 + 1). These constants are called *literals* and fall into three classes: nonnumeric (alphameric), numeric, and floating-point.

Nonnumeric literals are always bounded by single quotation marks and may consist of any members of the EBCDIC character set except the quotation mark. A nonnumeric literal (which could contain only numeric characters) may be from 1 through 120 characters in length, thus giving rise to the need for continuation card capability. When part of a nonnumeric literal must be carried over to the following card, a hyphen is placed in column 7 of the continuation line and a quotation mark is placed in Margin B preceding the continuation of the literal. All spaces at the end of the continued line and any spaces following the quotation mark in the continuation line and preceding the final quotation mark of the literal are considered part of the literal.

Examples of nonnumeric literals are:

```
'CARD 3 MISSING'
'PART 817A-23 OUT OF STOCK'
'SEPTEMBER 30, 1967'
'5428'
```

Numeric literals contain from 1 through 18 digits, with preceding plus or minus sign optional, and with decimal point optional (except that the decimal point may not appear as the rightmost character). Numeric literals which do not contain decimal points are assumed to be integers.

Examples of numeric literals are:

```
1
+803.43
-1.000625
.000005
134290062428
```

Floating-point literals are written as numeric literals of no more than 16 digits and with required decimal point, followed by the letter E and a one or two-digit signed or unsigned exponent. As with Assembler Language and FORTRAN, the magnitude of such a number may not exceed $.72 \times 10^{76}$.

Examples of both floating-point literals and their equivalents are

Literal	*Equivalent*
3.14159E0	3.14159
832.42E-6	.00083242
-3.0012E+3	-3001.2
4.E00	4
.000005925E12	5925000

15.2 STORAGE ALLOCATION FOR NUMERIC VARIABLES

The COBOL programmer may reserve storage areas for numerical values under assigned names by specifications in the DATA DIVISION. A detailed description of each variable's format can be given by means of the PICTURE specification and the method of storage can be defined by the COMPUTATIONAL specification.

15.2.1 The PICTURE Specification

The PICTURE specification allows the COBOL programmer to provide a digit-by-digit description for an assigned storage area. For numerical variables, each digit is denoted by a 9. Thus a description of a five-digit unsigned integer would be

```
PICTURE 99999
```

or, alternatively,

```
PICTURE IS 99999
```

If a sign is to be included, the PICTURE specification is started with an S. When a given variable is to contain a mixed number, the location of the decimal point is designated by a V. For example, if we wish to store a value in the form ±XXXX.XX our specification would be

```
PICTURE S9999V99
```

Nonnumeric values (i.e., those to be stored as bytes) are described using an X for each character. Thus a five-character field named B is described as

```
B PICTURE XXXXX
```

A shorthand form can be used in which the number of characters is indicated in parentheses. The above specification, for example, may also be written as

```
B PICTURE X(5)
```

15.2.2 The COMPUTATIONAL Specification

With the required amount of storage being specified by the PICTURE specification, we can define the method of storage by the COMPUTATIONAL specification. The various options are summarized in Table 15.1.

As an illustration, let us allocate storage for a variable named C4 which

TABLE 15.1 USE OF COMPUTATIONAL OPTIONS

Option	Internal Format	Number of Specified Digits in PICTURE	Number of Bytes
COMPUTATIONAL	Fixed-Point Binary	1-4	2
		5-9	4
		10-18	8
COMPUTATIONAL-1	Floating-Point Short Operand	1-4	2
		5-9	4
		10-18	8
COMPUTATIONAL-2	Floating-Point Long Operand	1-4	2
		5-9	4
		10-18	8
COMPUTATIONAL-3	Packed-Decimal	1	1
		2	2
		3	2
		4	3
		5	3
		6	4
		7	4
		8	5
		9	5
		10	6

we wish to store in packed decimal format and to be treated as ±XXX.XXXX.

Our specification would be

```
77  C4 PICTURE S999V9999  COMPUTATIONAL-3.
```

The period after the 3 is required. We could be somewhat more readable and say

```
77  C4 PICTURE S999V9999  USAGE IS COMPUTATIONAL-3
```

15.2.3 Assignment of Initial Values

Specifications of numerical variables may be supplemented by initial value assignments. The general form consists of the alternative specifications

```
VALUE
```

or

```
VALUE IS
```

followed by the desired value. Thus the specification

```
77  CC2 PICTURE S99V9  COMPUTATIONAL-3 VALUE IS 17.9.
```

will store 17.9 in packed-decimal form under the name CC2.

15.3 BASIC COMPUTATIONAL KEYWORDS

As was the case with previously discussed languages, we shall introduce various COBOL operations more or less arbitrarily and reinforce the concepts with examples.

15.3.1 Addition and Subtraction

Addition in COBOL is specified in several ways.

1. ADD *v1 v2 v3* . . . GIVING *result*.
   v1 v2 v3 . . . represent the names of the addends, separated by one or more blanks. Thus if we write

   ```
   ADD X Y Z GIVING W.
   ```

   the value equivalent to X + Y + Z will be placed in W, replacing its previous contents. Any member of the addend list could be a literal. For instance, the statement

   ```
   ADD X 17.4 2.64E2 Y GIVING Z.
   ```

   will place a value equivalent to X + Y + 17.4 + 264 in Z.
   This list must contain at least two items.
2. The keyword GIVING may be replaced by TO, making the general form

   ```
   ADD v1 v2 v3 . . . TO v.
   ```

   When this form is used, the current value of *v* is increased by the sum of the items between ADD and TO. As before, any of these items (except *v*) may be a name or a literal.
   A typical example would be

   ```
   ADD 1 TO X.
   ```

3. Another way to specify addition is by means of the COMPUTE verb, in which case the form is

   ```
   COMPUTE result = v1 + v2 + v3 + · · ·
   ```

15.3.2 The ROUNDED Option

Rounding may be specified by including the keyword ROUNDED following the variable name for the result.

```
ADD X 13.9 Y GIVING X ROUNDED
```

When included, rounding to the low-order position of the result takes place, according to its PICTURE. When not included and when the result as com-

puted occupies more low-order places than the result field as specified in its PICTURE, truncation occurs. For instance, if X is 17.40, Y is 6.31, and the PICTURE for Z is 99V9, the statement

```
COMPUTE Z = X + Y + 32.77.
```

will place a value of 56.4 in Z, while

```
COMPUTE X ROUNDED = X + Y + 32.77.
```

will produce a value of 56.5.

Related to ROUNDED is the SIZE ERROR option. As ROUNDED applies to the low-order digits to the right of the decimal point, SIZE ERROR applies to the number of integral places in the result. It provides a convenient way to specify what special processing is desired when the number of integral places in a calculated result is greater than the space provided via the PICTURE of the result. When the SIZE ERROR option is used as part of an ADD statement, either ON SIZE ERROR or SIZE ERROR is followed by a statement specifying an unconditional action to be taken by the object program, in exactly the way a statement is used on the right side of the Logical IF statement in FORTRAN. For example, consider the following statement:

```
ADD FACTOR TO SUM ON SIZE ERROR GO TO CORRECTION.
```

Here if the integral part of the result is too great to be contained by SUM, control is sent to a procedure named (by the problem programmer) CORRECTION by the COBOL verb GO TO. If the SIZE ERROR option has not been exercised, and if a size error condition arises, processing continues; the numerical value of the result, however, is likely to be in error.

The SUBTRACT verb is used in a similar way. Its basic form is

```
SUBTRACT v1 FROM v2 GIVING v3.
```

In this case *v*3 will contain *v*2 − *v*1. If we were to say

```
SUBTRACT X Y Z FROM W.
```

then W's value would be reduced by (X+Y+Z). The GIVING clause can be used with this form as well. COMPUTE, ROUNDED, and ON SIZE ERROR options may also be used as with addition.

15.3.3 The DISPLAY Verb

DISPLAY is a COBOL verb that provides a primitive output capability similar (though superior) to the PDUMP subprogram capability in FORTRAN. DISPLAY is always followed by the name(s) of data and/or literals. If the statement concludes with UPON CONSOLE or UPON SYSPUNCH, output is produced on the console device or on the system punch, respectively; if the statement does not contain this additional information, output

is produced on whatever device has been specified as SYSOUT, the system-output device. For the first two options, a maximum of 72 characters may be displayed each time DISPLAY is used; for the third option, the limit depends on the device used as SYSOUT.

Output produced by the DISPLAY statement is automatically converted, if required, from internal format to external format. Packed-decimal and fixed-point binary fields are converted to zoned-decimal format, and floating-point fields are converted to the external floating-point format used in floating-point literals. The minus sign of negative fields is combined with the low-order digit into a single alphabetic character according to the code used in punching cards.

EXAMPLE 77

Write a program to add the factors F1, F2, and F3, each of which is known to two decimal places, to round the result to the nearest integer value, and to display the result at the System 360 console. Assume the factors are in memory in packed-decimal format and may have up to four digits to the left of the decimal point.

Solution Program

Source Program Statements (*columns* 8–72)

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'EXAMPLE77'.
AUTHOR. I. M. RUFUS
INSTALLATION. NEW JERSEY HEADQUARTERS
DATE-WRITTEN. JUNE 14, 1968
DATE-COMPILED. JUNE 22, 1968
SECURITY. CODE A-2
REMARKS. COBOL DOES NOT CONTAIN THE SAME KIND OF COM-
    MENTS CARD CAPABILITY AS FORTRAN. INSTEAD, THIS MULTI-
    LINE REMARKS CAPABILITY IS PROVIDED IN THIS DIVISION
    AND THE NOTE VERB IS PROVIDED FOR USE IN THE PROCE-
    DURE DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360 F65.
OBJECT-COMPUTER. IBM-360 F50.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  F1
    PICTURE S9999V99 COMPUTATIONAL-3
    VALUE IS 1312.58.
77  F2 PICTURE S9999V99 COMPUTATIONAL-3 VALUE-406.23.
77  F3
    PICTURE S9999V99 COMPUTATIONAL-3
    VALUE 4098.88.
77  SUM
    PICTURE S99999 COMPUTATIONAL-3.
```

```
PROCEDURE DIVISION.
    ADD F1 F2 F3 GIVING SUM ROUNDED.
DISPLAY SUM UPON CONSOLE.
STOP RUN.
```

The size of this Solution Program appears all out of proportion to the required processing. There are two reasons for this. First, COBOL features beyond the bare minimum required to carry out the required processing are illustrated; second, the structure of COBOL was not designed for illustrative examples such as this but for larger and more realistic problems.

The full range of alternatives is illustrated in the IDENTIFICATION DIVISION. After each of the reserved COBOL terms (AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, SECURITY, and REMARKS), problem programmer entries which must be contained within Margin B are appended. These entries may utilize any of the characters in the EBCDIC set. On the surface, there appears to be no reason why these identification lines should be treated in a different way from the line containing PROGRAM-ID followed by the program name. The significant difference is that the program name is used by the control program in storing and retrieving the object program.

15.3.4 The ACCEPT Verb

ACCEPT is a verb that provides the capability of reading a single field into memory, when the console is specified, as in the statement

```
ACCEPT X FROM CONSOLE.
```

The input device is the console and the field must not exceed 72 characters in length. If the simpler form

```
ACCEPT Y1
```

is used, the input device is the assigned system input device (SYSIN) and the field length is determined by its PICTURE description, within the limits of the device being used.

15.3.5 The MOVE Verb

Internal assignments in COBOL are handled with the MOVE verb. For example,

```
MOVE X TO Y.
```

sets the contents of Y equal to those of X. The field being moved (X in this case) could be a literal as well as a variable.

In general, the field being moved is called the *source* field, and the field which the source field replaces is called the *receiving* field. If USAGE is defined

differently for the source and the receiving fields. MOVE not only moves the source field but converts it to the USAGE of the receiving field. If the PICTURE of the source field differs from that of the receiving field, data may be edited while being moved. Further note of this feature will be made when the PICTURE clause is more fully described.

15.4 SEQUENCE CHANGING IN COBOL

As is the case in FORTRAN, a number of fairly simple mechanisms is available for specifying a wide range of decision rules and branches.

15.4.1 The GO TO Statement

An unconditional branch is specified by a statement of the form

GO TO *name.*

where name refers to some procedure name (Sec. 15.1.3).

COBOL also has the equivalent of FORTRAN's computed GO TO statement.

For example, the statement

GO TO P-1 P-2 P-3 P-4 DEPENDING ON COUNT.

sends control to procedures P-1, P-2, P-3, or P-4, depending on whether the value of the variable COUNT is 1, 2, 3, or 4, respectively. The variable on which the transfer depends (COUNT in this case) must be an integer no longer than four digits and must have USAGE specified as DISPLAY, COMPUTATIONAL, or COMPUTATIONAL-3. If this variable is larger than the number of procedure names in the statement, control passes to the next consecutive statement.

COBOL also contains the facilities for constructing switches (equivalent of the FORTRAN ASSIGN/Assigned GO TO combination). This is done via the ALTER/GO TO combination and may be used whenever an unconditional GO TO, with or without a subsequent procedure name, is the only statement in a paragraph that has a paragraph name. For example, consider the following statements of a program;

ALTER REGULAR TO PROCEED TO OVERTIME.

```
REGULAR. GO TO DOUBLE-TIME.

.

ALTER REGULAR TO PROCEED TO DOUBLE-TIME.
```

Whenever the first ALTER statement is executed, the GO TO is altered, in effect, to GO TO OVERTIME: whenever the second ALTER statement is executed, the GO TO operates as written, GO TO DOUBLE-TIME.

15.4.2 The IF Statement

The IF statement has the general form

IF *condition* THEN *statements* ELSE *alternates*.

When the condition specified immediately following the keyword IF is true, the statement(s) following THEN is(are) executed. Should that condition turn out to be false, the statements following THEN are ignored and those following ELSE (specified by *alternates*) are executed. OTHERWISE may be used as an alternate for ELSE, and NEXT SENTENCE may be used as an alternate for either set of statements.

The condition part of an IF statement may be of several forms, called *relation test*, *sign test*, *class test*, *condition name test*, and *overflow test*.

A relation test is of the form, (*an arithmetic expression*) (IS) (or IS NOT) (GREATER THAN, LESS THAN, or EQUAL TO) (*another arithmetic expression*). GREATER THAN may be written > ; LESS THAN may be written < ; and EQUAL TO may be written = . An example of a relation test is:

```
ON-HAND + ON-ORDER IS GREATER THAN 3*DEMAND
```

A sign test is of the form, (*an arithmetic expression*) (IS) (or IS NOT) (POSITIVE, ZERO, or NEGATIVE). For example, the condition

```
X IS NEGATIVE
```

is a sign test.

A class test is of the form, (*a field*) (IS) (or IS NOT) (NUMERIC or ALPHABETIC). The condition, FIELD1 IS NOT NUMERIC, is such a test. Only packed-decimal, zoned-decimal, and alphameric fields may be so tested.

A condition name test is similar to a relation test except that the relation is incorporated as part of the variable name through a special usage in the DATA DIVISION discussed with Example 81. For example, the statement, IF OVER-21 GO TO MAJOR ELSE GO TO MINOR, might be used in connection

with the DATA DIVISION, in which the value of OVER-21 is set to a particular code, say 1.

An overflow test, unlike the four preceding tests, has a single purpose: to test for an end-of-page (overflow) condition when an output file is being printed. If a variable name, say FILE-10, is suitably defined for this purpose in the ENVIRONMENT DIVISION, a statement such as IF FILE-10 GO TO END-OF-PAGE-PROCEDURE may then be used in the PROCEDURE DIVISION to initiate a special procedure when the page-overflow condition occurs.

In addition to the above five types of simple conditions that may be used in constructing IF statements, a virtually unlimited variety of decision-making capabilities may be specified in two ways. First, compound conditions may be formed by connecting simple conditions via logical operators; and second, IF statements may be "nested" within IF statements.

The formation of compound conditions is carried out using the logical operators AND, OR, and/or NOT, where these operators are defined as they were in FORTRAN. For example, the simple conditions AGE IS GREATER THAN 65 and EMPLOYMENT-CODE = 1 may be combined into a compound condition such as AGE IS GREATER THAN 65 OR EMPLOYMENT-CODE = 1, which is true if either of its simple conditions is true. Where compound conditions form hierarchies of simple and compound conditions, parentheses may be used as with arithmetic expressions to remove possible ambiguities from the meaning of the resultant expression. Specifically, the above compound condition might be written

(AGE IS GREATER THAN 65) OR (EMPLOYMENT-CODE = 1)

To consider how IF statements may be nested within IF statements, look again at the general form:

IF *condition* THEN *statements* ELSE *alternates*.

If either or both of these sets of statements to be executed includes another IF statement, then the IF statements are said to be *nested*. An illustration of this feature is given in flowchart form in Figure 15.2 and also by means of the COBOL statements as following: IF AGE IS GREATER THAN 65 MOVE CODE-A TO WORK-AREA IF SOCIAL SECURITY = 8 GO TO SS65 ELSE GO TO NSS ELSE ADD EARNINGS TO BALANCE MOVE CODE-A TO TABLE-AREA GO TO PL-RECORD. As with nested DO loops in FORTRAN, notice how the internal IF statement is completed prior to continuation at the next outer level.

EXAMPLE 78

Write a program to read a floating-point number from the console device. If its value is less than or equal to 100, write NOT GREATER THAN 100 on the console device and terminate execution of the program. If its value is greater than 100, write DATUM WITHIN RANGE on the console device and pause. After processing

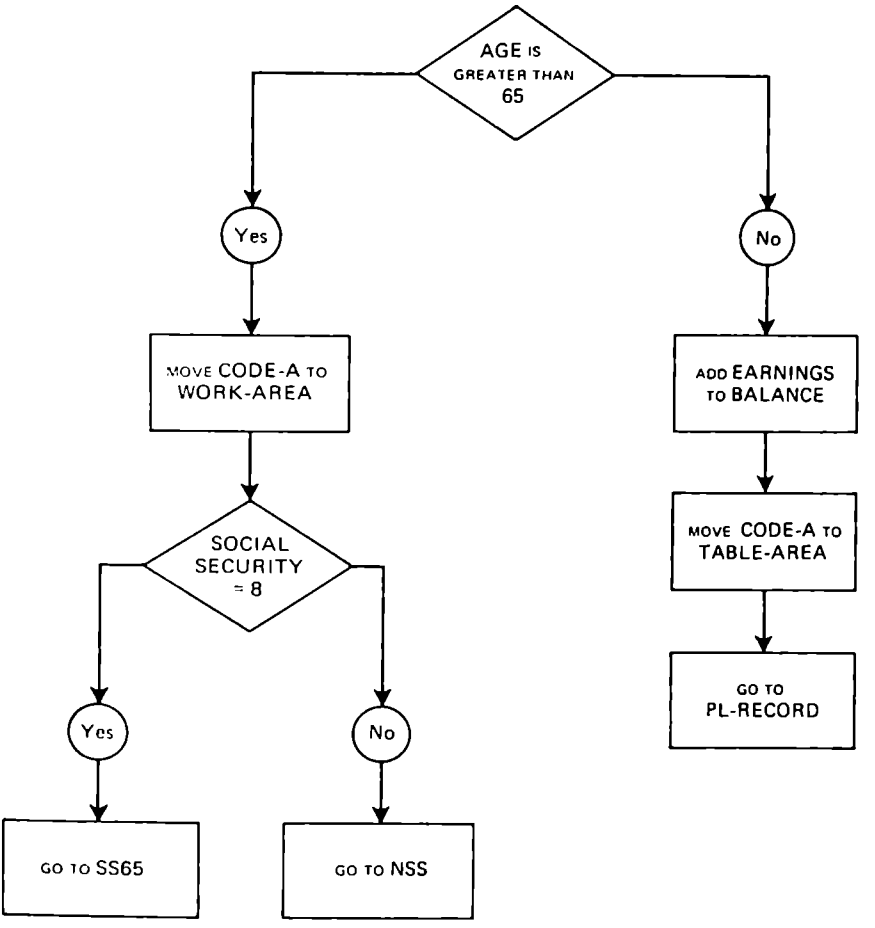


Figure 15.2 Illustration of nested IF statements

is restarted by the operator, compute the square root of the number and write it on the console device.

Solution Procedure

Use the ACCEPT verb for input and the IF statement for the test. The DISPLAY verb will print the first message, and the STOP verb will handle the second, following which the program will pause. Newton's method will be used to compute the square root.

Solution Program

Source Program Statements (columns 8-72)

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'EXAMPLE78'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
```

Source Program Statements (columns 8-72)

```
SOURCE-COMPUTER. IBM-360 F65.
OBJECT-COMPUTER. IBM-360 E50.
DATA DIVISION.
WORKING STORAGE SECTION.
77  DATUM PICTURE -99999V9E-99 USAGE IS DISPLAY.
77  RADICAND USAGE IS COMPUTATIONAL-1.
77  APPROX USAGE IS COMPUTATIONAL-1.
77  ROOT USAGE IS COMPUTATIONAL-1.
77  TEST USAGE IS COMPUTATIONAL-1.
PROCEDURE DIVISION.
    ACCEPT DATUM FROM CONSOLE.
    IF DATUM IS GREATER THAN 100 GO TO M-2 ELSE
    GO TO M-1.
M-1. DISPLAY 'NOT GREATER THAN 100' UPON CONSOLE.
    STOP RUN.
M-2. STOP 'DATUM WITHIN RANGE'.
    MOVE DATUM TO RADICAND.
    MOVE DATUM TO APPROX.
LOOP. COMPUTE ROOT = .5 * (APPROX - RADICAND/APPROX).
    IF ROOT-APPROX IS NEGATIVE MOVE APPROX-ROOT TO TEST
    GO TO ACCURACY ELSE
    COMPUTE TEST = ROOT-APPROX GO TO ACCURACY.
ACCURACY. IF TEST - .00001 IS POSITIVE MOVE ROOT TO APPROX
    GO TO LOOP ELSE NEXT SENTENCE.
    DISPLAY ROOT UPON CONSOLE.
    STOP RUN.
```

In the WORKING STORAGE SECTION of the Solution Program are the definitions of DATUM, an external (to memory) floating-point number, and of APPROX, ROOT, and TEST, three internal, floating-point fields. Because DATUM may be of arbitrary length and may have the decimal point in a number of positions, a PICTURE is required. USAGE, however, IS DISPLAY as it is for all external fields and for all internal fields in which each character occupies one byte. Because APPROX, ROOT, and TEST are defined as COMPUTATIONAL-1 (short-precision floating-point and therefore fixed-format), no PICTURE is required. When USAGE is not specified, COBOL assumes DISPLAY.

The PICTURE of an external floating-point field such as DATUM always consists of (1) a + or -, (2) one to sixteen 9's representing digit positions and with a leading, embedded, or trailing decimal point or V (representing the position of the decimal point), (3) the letter E, (4) a + or -, and (5) two 9's representing the exponent. The plus and minus signs are used in the following way: Plus indicates that a plus sign is used with positive quantities and a minus sign with negative quantities; minus indicates that a blank is used for positive values and a minus sign for negative values.

The statement, DISPLAY 'NOT GREATER THAN 100' UPON CONSOLE, is like the statement, DISPLAY SUM UPON CONSOLE, of the previous example except that

the quantity to be displayed is expressed as a literal rather than as the name of a variable. The statement, STOP 'DATUM WITHIN RANGE', however, is quite a bit different from the previously encountered STOP RUN statement. STOP followed by a literal produces the following results:

1. Both a compiler-generated message code and the specified literal are displayed upon the console.
2. The object program pauses (stops in a restartable condition).
3. The program may be restarted only by entering the above message code via the console.

15.5 BASIC INPUT/OUTPUT SPECIFICATIONS

When a program contains one or more input and/or output data files, there is an effect on the ENVIRONMENT, DATA, and PROCEDURE DIVISIONS. The ENVIRONMENT DIVISION is generally augmented with an INPUT-OUTPUT SECTION, a FILE CONTROL paragraph, and a SELECT sentence with ASSIGN and RESERVE clauses. In the DATA DIVISION, there is a FILE SECTION with FD, file name, RECORDING, RECORD, LABEL RECORDS, and DATA RECORD clauses, and also a set of entries which describe both the record and its contents on a level-by-level basis. Finally, in the PROCEDURE DIVISION, the effect of the data file is the use of the OPEN, READ, and END verbs.

15.5.1 The FILE CONTROL Paragraph

COBOL is informed that a particular file is to be used by the inclusion of a FILE CONTROL paragraph in the ENVIRONMENT DIVISION. The basic format is

```
INPUT-OUTPUT SECTION.
FILE CONTROL.
  SELECT filename ASSIGN clause
  RESERVE clause
```

15.5.1.1 The SELECT and ASSIGN Keywords

The SELECT . . . ASSIGN construction in the FILE CONTROL paragraph is used to link the name of the data file as chosen by the problem programmer to the name of the file as known to the operating system's control program and to the class and number of the input/output device. The three classes of input/output devices and the specific units that belong to these classes are as follows:

DIRECT-ACCESS	7320, 2301, 2302, 2311, 2321, 2314
UNIT-RECORD	1442, 1403, 1404, 1443, 1445, 2501, 2520, 2540
UTILITY	2400, 7320, 7340, 2301, 2302, 2311, 2321, 2314

In general, the SELECT sentence with its associated clauses must be given for each file to which reference is made in the COBOL source program. For example, the assignment of a file name BRINGIN to the 1442 card reader with an installation designation of F001 is as follows:

```
INPUT-OUTPUT SECTION.
FILE CONTROL.
  SELECT BRINGIN ASSIGN TO F001 UNIT-RECORD 1442 UNIT.
```

15.5.1.2 The RESERVE Clause

The RESERVE NO ALTERNATE AREAS clause is an option that may be specified with respect to buffer areas. The alternate option is of the form, RESERVE *n* ALTERNATE AREA(S), where *n* is an integer in the range 1–254. In either case, this clause refers to input/output buffers in addition to the standard minimum of one required for each file.

15.5.1.3 The ACCESS Clause

The manner in which records are to be read from a file is specified by the ACCESS clause, which reads

```
ACCESS IS SEQUENTIAL
```

or

```
ACCESS IS RANDOM
```

SEQUENTIAL is automatically assumed when the ACCESS clause is omitted.

Other clauses, not yet illustrated, that may be used within the FILE CONTROL paragraph are ORGANIZATION, SYMBOLIC KEY, ACTUAL KEY, RECORD KEY, TRACK-AREA, and FILE-LIMIT. In addition, the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION may include an I-O-CONTROL paragraph with SAME, RERUN, and/or APPLY clauses.

15.5.2 The FILE SECTION

The FILE SECTION of the DATA DIVISION is used to describe the characteristics of files and the way in which the associated input/output areas are organized. In general (but this statement will be altered in a subsequent chapter), each file described in this section starts with FD followed by the name of the file, then contains a number of clauses.

15.5.2.1 The RECORDING MODE Clause

RECORDING MODE refers to record lengths as F (fixed-length), V (variable-length), or U (unspecified). Format V logical records each contain a preceding control word specifying their size; format U logical records, which also may be of variable length, do not contain this control word.

15.5.2.2 The BLOCK CONTAINS Clause

Where a physical record consists of a block of logical records (not illustrated in this Solution Program), the clause, BLOCK CONTAINS *n* RECORDS, is used, *n* being an integer.

15.5.2.3 The RECORD CONTAINS Clause

This clause specifies the number of characters in a file record. For fixed length records, the form is

RECORD CONTAINS *n* CHARACTERS

When a variable length record is being used the more extended form is

RECORD CONTAINS *m* TO *n* CHARACTERS

where m is an integer $< n$.

15.5.2.4 The LABEL RECORDS Clause

Labels, ordinarily used with magnetic tape or disk files, may be constructed according to standard specifications set by the operating system, according to the programmer's own design, or they may be omitted. Pertinent information is given in the LABEL RECORDS clause. The available forms are

LABEL RECORDS ARE STANDARD
LABEL RECORDS ARE OMITTED (for UNIT-RECORD files)
LABEL RECORDS ARE *name*

In the last mentioned option, name refers to a programmer-supplied routine for handling nonstandard labels. No files read from or written on unit record devices ever have label records. A tape file, on the other hand, may have labels omitted, standard labels, or a combination of standard and programmer-devised labels. Files with indexed, direct, or relative organization must *always* have STANDARD labels.

15.5.2.5 The DATA RECORD Clause

When a file is organized as a collection of records (which is the usual practice for COBOL), the DATA RECORD clause is used to assign a collective name to all the items in a record. The general form is

DATA RECORD IS *recname*

As an example, let us set up a FILE SECTION for a file named PAYMSTR. This file, containing fixed-length records, is stored on a tape which is

mounted on a 2400 unit. Each record consists of 200 characters and the records are stored in groups of five. Standard labeling is used. If we wish to refer to an entire record (all 200 characters) by the collective name ONEREC, our complete specifications would read

```
FILE SECTION.
FD PAYMSTR
   RECORDING MODE IS F RECORD CONTAINS 200
   CHARACTERS LABEL RECORDS ARE STANDARD DATA
   RECORD IS ONEREC.
```

15.5.2.6 Description of Components in a Data Record

Once the modifying clauses have been written, the contents of a file are specified according to the format of the logical records. A system of hierarchical level numbers (level number 77 has already been encountered in the WORKING STORAGE SECTION of this Division) is used to define the relationship of fields to subfields within them, to fields within the subfields, etc. Logical records have level number 01, and consecutive level numbers through level number 49 may be used to describe these relationships within each record. Level number 77 is a special one, referring only to independent (not related to a file) fields in the WORKING-STORAGE SECTION and to an entry in the not yet encountered LINKAGE SECTION. A second special level number is level number 88, which is used to define a condition name for a related conditional variable.

Each logical record is assigned level number 01 (the leading zero may be replaced by a blank in this and in other one-digit level numbers, if desired) and the fields within it are assigned the subordinate level number 02 or higher. If a subfield has no subordinate field, PICTURE and USAGE entries are made with it, in addition to its name. This system of record specification facilitates the handling in the PROCEDURE DIVISION of data groups appropriate to the required processing.

To illustrate this hierarchical data organization, let us consider the following:

```
01  PERSONNEL-RECORD.
  02  NAME.
    03  FIRST-NAME PICTURE X(10) DISPLAY.
    03  MIDDLE-NAME PICTURE X(10) DISPLAY.
    03  LAST-NAME PICTURE X(15) DISPLAY.
  02  ADDRESS.
    03  NUMBER PICTURE XXXXX DISPLAY.
    03  STREET PICTURE X(10) DISPLAY.
    03  CITY PICTURE X(10) DISPLAY.
    03  ZIP PICTURE XXXXX DISPLAY.
```

```
02  BIRTHDATE.
    03  MONTH PICTURE 99 COMPUTATIONAL-3.
    03  DAY PICTURE 99 COMPUTATIONAL-3.
    03  YEAR PICTURE 99 COMPUTATIONAL-3.
02  DEPT PICTURE X(5) DISPLAY.
02  JOB-CODE PICTURE XX DISPLAY.
02  HOURLY.
    03  STRAIGHTTIME PICTURE 9V99 COMPUTATIONAL-3.
    03  OVERTIME PICTURE 9V99 COMPUTATIONAL-3.
```

The name PERSONNEL-RECORD refers to all of the items which follow. By the same token, each level 02 name is a collective reference for all fields subordinate to it. Thus a specified action in which ADDRESS is an operand will cause that action to be performed on NUMBER, STREET, CITY, and ZIP.

While level numbers were assigned consecutively in this example, this is not required; only the relative sizes are significant. In the above example the level numbers were written in the optional indented format; they may, however, be written one under the other with no indentation in margin A.

15.6 THE OPEN AND CLOSE VERBS

The OPEN statement makes a file available for processing, while the CLOSE statement terminates processing for a file. (In this respect, these are analogous to the OPEN and CLOSE macro instructions in Assembler Language.)

15.6.1 Opening Files

A file in COBOL is either INPUT, OUTPUT, or I-O. One of these three types must be specified after the verb OPEN which, in turn, is followed by the file name. For example:

```
OPEN INPUT PAYMSTR.
```

makes the file named PAYMSTR available as an input file. Several files may be opened with one statement. In the illustration shown here,

```
OPEN INPUT FILE1 OUTPUT FILE2 FILE3 I-O FILE4.
```

an INPUT file FILE1, two OUTPUT files FILE2 and FILE3 and an I-O file FILE4 are opened.

When a file is on magnetic tape, it is automatically rewound upon opening. If this is undesirable, it can be prevented by specifying

```
WITH NO REWIND or simply NO REWIND
```

after the file name. (Rewinding positions the tape at the first data record, past the label, if there is one). Thus, the statement

OPEN OUTPUT FILE1 FILE2 WITH NO REWIND

will open and rewind FILE1 and will open FILE2 without rewinding.

In addition to the above option, the INPUT clause may also include the REVERSED option for specific files. This is used only for files which may be read backward (such as those stored on magnetic tape) and, of course, only when reading in the reverse of the normal direction is desired. A file may be opened as INPUT, OUTPUT, and/or I-O within the same program if there are intervening CLOSE statements and unless it has both indexed organization and sequential access.

15.6.2 Closing Files

The simplest CLOSE statement is merely

CLOSE *filename filename*

Repositioning to the beginning of the file takes place unless the REVERSED option was specified when the file was opened, in which case repositioning to the end of the file takes place. Each file named in the CLOSE statement may be followed by either one or neither of two options. In the first, either UNIT or REEL may be specified; in the second, either WITH NO REWIND (or just NO REWIND) or WITH LOCK (or just LOCK) may be specified.

The UNIT or REEL option is provided for use with disk units or magnetic tape reels, where the size of the data file may require more than one demountable unit or reel. It is used to institute the volume-switching procedures included in the operating system and applies only to files with standard sequential organization.

If neither UNIT nor REEL is specified, WITH NO REWIND causes positioning to the end of the file being processed (but if the REVERSED option has been exercised in the OPEN statement, positioning is to the beginning of the file). If neither UNIT nor REEL is specified, WITH LOCK causes the action (rewind, no rewind, lock) to be taken according to the DISP subparameter of the DD control card, external to the problem program.

15.7 THE BASIC READ STATEMENT

The form of the READ statement used for files with sequential access is

READ *filename* RECORD ATEND. . . .

This is sufficient to bring the next logical record from the named file into the storage area defined by the record description (Sec. 15.5.2). This form

has an optional INTO clause which, in effect, makes of the READ statement the equivalent of the above form and a MOVE statement. Thus the statement

```
READ PAYROLL RECORD INTO SALARIED AT END GO TO DISTRI-
BUTION.
```

both reads a record of the PAYROLL file into the input area and moves it to the field named SALARIED. The field to which such a move is made must have been defined in the WORKING STORAGE SECTION or must be a previously opened output record or must be a saved-area record (to be described in Chapter 17).

15.8 FIGURATIVE CONSTANTS

COBOL has several keywords which represent names for specific values that are frequently used in data processing operations. Such names, called *figurative constants*, facilitate the use of narrative statements consistent with the overall tenor of the language.

All figurative constants except ZERO (and its equivalent forms) are always considered nonnumeric, but bounding quotation marks are never used with them. The list of figurative constants follows, in all cases the singular and plural forms being equivalent and interchangeable.

Figurative Constants	*Used to Represent*
ZERO ZERO ZEROES	One or more zeros.
SPACE SPACES	One or more blanks or spaces.
HIGH-VALUE HIGH-VALUES	One or more appearances of the highest possible value (as used in sorting). A one-character field whose value is HIGH-VALUE may be used to operate as a delete-code or dummy-record-code character in files with indexed data organization and as a dummy-record-code character in files with relative organization.
LOW-VALUE LOW-VALUES	One or more appearances of the lowest possible value (as used in sorting).
ALL 'character'	One or more occurrences of the single character bounded by quotation marks. This character may not itself be a quotation mark.
QUOTE QUOTES	The quotation-mark character '. Note, however, that this figurative constant has no relation to the use of the quotation mark to bound nonnumeric literals.

For example, the statements

```
MOVE ZEROS TO COUNTER.
MOVE ALL 'B' TO WORD.
```

will set the field named COUNTER to zero and will fill WORD with as many B's as it takes.

15.9 THE EXAMINE VERB

EXAMINE statements are used to inspect a given field for the occurrence of a specified character. The number of occurrences may be determined and used in subsequent operations. In addition, it is possible to specify character substitution.

15.9.1 The TALLYING Form

When the number of occurrences of a certain character is to be counted, the EXAMINE statement takes the basic form

```
EXAMINE v1 TALLYING option 'a'
```

where *v1* is a programmer-defined variable whose USAGE must be DISPLAY (in other words, it must be stored one character per byte). *'a'* is any literal character recognized by COBOL or a figurative constant. COBOL sets up and uses a specific field called TALLY in which is stored a binary number equal to the number of occurrences of character *'a'* in field *v1*. *option* is one of three keywords which dictates the type of tallying that is done.

For example,

```
EXAMINE v1 TALLYING ALL 'a'
```

will cause all occurrences of *'a'* in *v1* to be counted, as expected.

Alternatively, if we use LEADING instead of ALL, *v1* is examined (from left to right) and a tally is made of all occurrences of *'a'* until COBOL encounters a character other than *'a'*.

The third option, UNTIL FIRST, operates somewhat differently. When this is specified, TALLY will contain the number of characters in *v1* to the left of (ahead of) the first *'a'*. In all cases, TALLY may be used as an operand in subsequent statements.

The TALLYING form of the EXAMINE statement may be extended by adding the REPLACING BY option:

```
EXAMINE v1 TALLYING option 'a' REPLACING BY 'b'.
```

where *'b'* is another character consistent with *v1* (*'b'* cannot be numeric if *v1* is alphabetic).

When ALL is used, all *'a'*s are replaced by *'b'*s. With the LEADING option, replacement of *'a'*s with *'b'*s occurs until the first character other than an

'a' is encountered. UNTIL FIRST and REPLACING BY causes all characters other than *'a'* to be replaced with *'b'*s until the first *'a'* is reached. In all cases, tallying is done as before.

15.9.2 The REPLACING Form

If tallying is not required, the EXAMINE statement can be used strictly for replacement by writing

EXAMINE *v1* REPLACING *option 'a'* BY *'b'*

The italicized symbols have the same meanings as before. In addition to three options described previously, a fourth option, FIRST, is also available with this form. If we say

EXAMINE *v1* REPLACING FIRST *'a'* BY *'b'*

only the first (leftmost) *'a'* encountered will be replaced.

EXAMPLE 79

A data file is punched on cards. Each card contains four six-digit integers in columns 1–6, 7–12, 13–18, and 19–24, an eight-digit integer in columns 25–32, and alphameric information in columns 33–50. For each card in which the sum of the six-digit integers does not equal the eight-digit integer, display 'KEYPUNCH ERROR' and columns 1–6 on the console. Tally the number of occurrences of the letter A in the alphameric fields of all the cards, and display 'END OF FILE' and this sum after all the cards have been read.

Solution Procedure

Because a data file is involved, the ENVIRONMENT DIVISION must include an INPUT-OUTPUT SECTION in addition to a CONFIGURATION SECTION. Also, the DATA DIVISION must include a FILE SECTION which defines the file in detail. Use the verbs OPEN, READ, and CLOSE in connection with reading the data file; use SUBTRACT to determine the difference between the sum of the first four integers and the fifth; and use EXAMINE to tally the occurrences of the letter A.

Solution Program

Source Program Statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'EXAMPLE79'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360 E50.
OBJECT-COMPUTER. IBM-360 E50.
INPUT-OUTPUT SECTION.
FILE CONTROL.
  SELECT INFILE ASSIGN TO F001 UNIT-RECORD 2540 UNIT
  RESERVE NO ALTERNATE AREAS.
```

Source Program Statements

```
DATA DIVISION.
FILE SECTION.
FD INFILE
   RECORDING MODE IS F RECORD CONTAINS 80 CHARACTERS
   LABEL RECORDS ARE OMITTED DATA RECORD IS DATA-XY.
01  DATA-XY
   02  I-1 PICTURE 999999 COMPUTATIONAL-3.
   02  I-2 PICTURE 999999 COMPUTATIONAL-3.
   02  I-3 PICTURE 9(6) COMPUTATIONAL-3.
   02  I-4 PICTURE 9(6) COMPUTATIONAL-3.
   02  I-5 PICTURE 9(8) COMPUTATIONAL-3.
   02  ALF PICTURE X(18) DISPLAY.
WORKING-STORAGE SECTION.
77  TOTAL PICTURE 999999 COMPUTATIONAL-3.
PROCEDURE DIVISION.
   OPEN INPUT INFILE.
   MOVE ZEROS TO TOTAL
LOOP. READ INFILE RECORD AT END GO TO END-PROCEDURE.
   SUBTRACT I-1 I-2 I-3 I-4 FROM I-5.
   IF I-5 IS NOT ZERO DISPLAY 'KEYPUNCH ERROR' I-1 ELSE NEXT SENTENCE.
   EXAMINE ALF TALLYING ALL 'A'.
   ADD TALLY TO TOTAL.
   GO TO LOOP.
END-PROCEDURE.
   DISPLAY 'END OF FILE. TOTAL IS' TOTAL.
   CLOSE INFILE.
   STOP RUN.
```

15.10 MULTIPLICATION AND DIVISION

As with addition and subtraction, COBOL allows multiplication and division operations to be specified in narrative or symbolic forms.

15.10.1 Multiplication

The MULTIPLY verb is used in statements of the form

```
MULTIPLY v1 BY v2 GIVING v3.
```

*v*1 and *v*2 may be variable names, numeric literals or floating-point literals while *v*3, of course, must be a variable name. The ROUNDED and SIZE ERROR options may be used as with addition or subtraction (Sec. 15.3.2).

Multiplication may also be performed using the COMPUTE form with the * operator. Thus

```
COMPUTE Y = X * Z.
```

will store the product of X times Z in Y.

15.10.2 Division

Division in COBOL is handled a little differently than might be expected. The basic narrative form is

DIVIDE *v1* INTO *v2* GIVING *v3*.

where *v1*, *v2* and *v3* have the same meanings as with MULTIPLY. As a result of this operation the *v2*/*v1* (*not* *v1*/*v2*) is stored in *v3*.

Using the slash (/) to denote division, we could also write

COMPUTE *v3* = *v2*/*v1*.

ROUNDED and SIZE ERROR options apply as before.

15.11 THE WRITE VERB

There are two general forms of the WRITE statement. When one specifies merely

WRITE *recname*.

COBOL writes the record currently in the output area (*recname*) onto the output file associated with recname. Recall that this association is established in the FILE SECTION of the DATA DIVISION, where *recname* is named in the DATA RECORD IS clause and described immediately thereafter. If we should wish to write onto an output file from some storage area other than that specifically reserved for output, we can say

WRITE *recname* FROM *area*.

in which case the output record named recname would be produced by writing the contents of *area*.

The second form of the WRITE statement is used for printed or punched output, and the first character in each logical record must be reserved by the programmer for use as a control character. This second form includes the WRITE *recname* and WRITE *recname* FROM *area* components of the first, then a clause: AFTER ADVANCING (*an integer or the name of a field*) LINES. If the integer variation of this clause is used, as in AFTER ADVANCING 2 LINES, the integer may be 0, 1, 2, or 3. These values designate skip to the next page, single space, double space, and triple space, respectively. If the field name variation of this clause is used, as in AFTER ADVANCING W LINES, the field named must be alphameric and one byte in length—that is, have the description PICTURE X. The possible values of this field and the interpretations of these values follow.

Value	*Interpretation*
blank	Single spacing
0	Double spacing
–	Triple spacing
+	Suppress spacing
1-9	Skip to channel 1-9, respectively
A,B,C	Skip to channel 10, 11, 12, respectively
V,W	Pocket select 1 or 2, respectively, on the IBM 1442, and 4 or 8 on the IBM 2540

By channel in the above table is meant the punching position on the paper carriage control tape of the printer.

EXAMPLE 80

A data file of several thousand records is punched into cards, one logical record per card. Columns 76-80 contain consecutive sequence numbers, and the file has been sorted accordingly. Write a program to convert this file to magnetic tape, blocking each 50 logical records into a physical record. Terminate processing if any cards are out of sequence. During the conversion, multiply the contents of columns 16-18 and 31-34 of each record and place the result in positions 50 56. Assume that the first of these fields is an integer and that the second and third have two places to the right of the decimal point. The decimal point is assumed, not punched.

Solution Procedure

With both an input file and an output file, two SELECT sentences are required, and two complete sets of file descriptions must be included in the FILE SECTION of the DATA DIVISION. While the description of the output file must contain a clause that specifies blocking each fifty logical records into a physical record, only logical records need be considered in the PROCEDURE DIVISION.

For sequence checking set a test variable to zero. Then for each record, check whether the sequence number less the test variable less 1 equals zero, and also replace the test variable with the current sequence number. Thus each sequence number will always be compared against the preceding sequence number. If the sequence numbers were sorted into ascending order, but not necessarily consecutive, the same approach could be used. In this case, however, only that the sequence number less the test variable was greater than zero would be checked.

Solution Program

Source Program Statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'EXAM80'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360 H65.
OBJECT-COMPUTER. IBM-360 H65.
INPUT-OUTPUT SECTION.
FILE CONTROL.
SELECT CARD-FILE ASSIGN TO F001 UNIT-RECORD 2540 UNIT
```

Source Program Statements

```
    ACCESS IS SEQUENTIAL RESERVE 1 ALTERNATE AREA.
SELECT TAPE-FILE ASSIGN TO F010 UTILITY 2400 UNIT
    ACCESS IS SEQUENTIAL RESERVE 1 ALTERNATE AREA.
DATA DIVISION.
FILE SECTION.
FD CARD-FILE
RECORDING MODE IS F
RECORD CONTAINS 80 CHARACTERS
LABEL RECORDS ARE OMITTED
DATA RECORD IS TRANSACTION.
01  TRANSACTION.
    02  FILLER PICTURE X(15).
    02  FIELD-A PICTURE 9(3).
    02  FILLER PICTURE X(12).
    02  FIELD-B PICTURE 99V99.
    02  FILLER PICTURE X(41).
    02  SEQUENCE PICTURE 9(5).
FD TAPE-FILE
RECORDING MODE IS F
BLOCK CONTAINS 50 RECORDS
RECORD CONTAINS 80 CHARACTERS
LABEL RECORD IS STANDARD
DATA RECORD IS T-FILE.
01  T-FILE.
    02  FILLER PICTURE X(49).
    02  PRODUCT PICTURE 9(5)V99 COMPUTATIONAL-3
    02  FILLER PICTURE X(24).
WORKING-STORAGE SECTION.
77  TEST PICTURE 9(5) USAGE COMPUTATIONAL-3.
PROCEDURE DIVISION.
    OPEN INPUT CARD-FILE OUTPUT  TAPE-FILE.
    MOVE ZEROS TO TEST.
LOOP. READ CARD-FILE RECORD INTO T-FILE
    AT END GO TO CLOSER.
    MULTIPLY FIELD-A BY FIELD-B GIVING PRODUCT.
    IF SEQUENCE-TEST-1 IS NOT EQUAL TO ZERO
    GO TO TERMINATE OTHERWISE NEXT SENTENCE.
    MOVE SEQUENCE TO TEST.
    WRITE T-FILE.
    GO TO LOOP.
TERMINATE. DISPLAY 'SEQUENCE ERROR'.
CLOSER. CLOSE CARD-FILE TAPE-FILE.
    STOP RUN.
```

Discussion

One of the clauses that is used to describe TAPE-FILE is: BLOCK CONTAINS 50 RECORDS. This clause was not required with the description of CARD-FILE because

the latter was not blocked: that is, there was just one logical record for each physical record. This clause enables the programmer, in the PROCEDURE DIVISION, to think as if a READ reads a single logical record and as if a WRITE writes a single logical record, regardless of blocking. The BLOCK CONTAINS clause must *never* be written for files with direct or relative organization, or when format U records are used.

The BLOCK CONTAINS clause may specify the number of characters, rather than the number of records, in a block. For example, BLOCK CONTAINS 4000 CHARACTERS is equivalent to BLOCK CONTAINS 50 RECORDS when each RECORD CONTAINS 80 CHARACTERS. An alternate form is: BLOCK CONTAINS 4000. For files with variable length (format V) records, the same form of the BLOCK CONTAINS clause may be used, but the number of characters specified must be calculated as the sum of the maximum number of data characters and four additional characters for each record. These additional characters are used to "remember" the specific sizes of the individual records.

In several of the record description entries for both data files in the Solution Program is apparently the same field name: FILLER. This word is not a field name but a standard COBOL term that may be used *in place of* a field name whenever no specific reference is to be made to such field in the PROCEDURE DIVISION. None of the entries using FILLER has a USAGE clause.

EXAMPLE 81

A master file which is stored on magnetic tape has 200-character records blocked 10 logical records to each physical record. These logical records are stored in ascending sequence, according to an 8-character index in positions 192–200 of each. Write a program to update this master file. Updating is based upon a deck of transaction cards which are sorted into the same sequence as the master file and which have their sorting indices punched in columns 1–8.

Updating is to be handled according to the following directions:

1. A master record for which there is no transaction record is retained unchanged.
2. A transaction record for which there is no master record generates an error message to the operator.
3. A transaction record with the digit 1 in column 9 indicates that the corresponding master record is to be deleted.
4. A transaction record with the digit 2 in column 9 indicates that columns 10–17 [PICTURE 9(6)V99] are to be divided into master record positions 60–69 [PICTURE 9(8)V99].
5. Transaction records for which there is a corresponding master record but in which the column 9 digit is other than 1 or 2 are to be printed in full, without editing. The master record is retained unchanged.

Solution Procedure

Once all the necessary definitions have been made and all four files have been opened, the first record is read from the transaction file and the first record is read from the original (old) master file. From this point the processing procedure splits into three distinct segments, depending on whether the sorting index of the master record is greater than, less than, or equal to that of the transaction record. Irre-

spective of the action taken, control is returned to this point again and again, until there are no more transaction records. At that point, remaining master records, if any are copied without change, the files are closed, and the processing is ended.

When the sorting index of a master record is greater than that of a transaction record, this indicates an error condition. The associated master record might be missing; the transaction card might be out of sequence; or the sorting index of the transaction card might be mispunched. According to the directions for this example, recognition of such an error condition, whatever the cause, should generate a message to the operator, that is, a DISPLAY (*some message*) UPON CONSOLE. Once this have been accomplished, the next transaction record is read and control is returned to the point at which the three-way split occurs.

When the sorting index of a master record is less than that of a transaction record, this indicates that no transaction record corresponding to that master record exists. In this case, the master record is to be retained unchanged, and copied, therefore, onto the new master file. The master record is written; the next master record is read; and control is returned to the point at which the three-way split occurs.

When the sorting index of a master record is equal to that of a transaction record, the master record is either deleted, altered, and retained, or it is retained without alteration with the transaction record being printed, depending on the digit in column 9 of the transaction card. A further three-way test, based on the digit in column 9, is made, and the appropriate action is taken. Deleting a master record is performed by reading the next record. Whichever of the three courses of action is taken, both the next transaction record and the next master record are read before control is returned to the point at which the first three-way split occurs.

Solution Program

Source Program Statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'EXAM81'.
REMARKS. EXCEPT FOR THE SPECIFIC MANNER IN WHICH THE
    TRANSACTION RECORDS ARE PROCESSED, THIS IS A TYPICAL
    EXAMPLE OF A FILE MAINTENANCE OPERATION ON A
    SEQUENTIAL FILE.
ENVIRONMENT DIVISION.
SOURCE-COMPUTER. IBM-360 F50.
OBJECT-COMPUTER. IBM-360 E50.
INPUT-OUTPUT SECTION.
FILE CONTROL.
SELECT OLD-MASTER ASSIGN TO F010 UTILITY 2400 UNIT
  ACCESS IS SEQUENTIAL RESERVE 1 ALTERNATE AREA.
SELECT NEW-MASTER ASSIGN TO F011 UTILITY 2400 UNIT
  ACCESS IS SEQUENTIAL RESERVE 1 ALTERNATE AREA.
SELECT TRANSACTION-FILE ASSIGN TO F001 UNIT-RECORD 2540 UNIT
  ACCESS IS SEQUENTIAL RESERVE 1 ALTERNATE AREA.
SELECT PRINT-FILE ASSIGN TO F003 UNIT-RECORD 1403 UNIT
  ACCESS IS SEQUENTIAL RESERVE 1 ALTERNATE AREA.
DATA DIVISION.
```

Source Program Statements

```
FILE SECTION.
FD OLD-MASTER
   RECORDING MODE IS F
   BLOCK CONTAINS 10 RECORDS
   RECORD CONTAINS 200 CHARACTERS
   LABEL RECORDS ARE STANDARD
   DATA RECORD IS OLD-MASTER-DATA.
01  OLD-MASTER-DATA.
   02  FILLER PICTURE X(59).
   02  FIELD-A PICTURE 9(8)V99 COMPUTATIONAL-3.
   02  FILLER PICTURE X(123).
   02  FIELD-B PICTURE X(8).
FD  NEW-MASTER
   RECORDING MODE IS F
   BLOCK CONTAINS 10 RECORDS
   RECORD CONTAINS 200 CHARACTERS
   LABEL RECORDS ARE STANDARD
   DATA RECORD IS NEW-MASTER-DATA.
01  NEW-MASTER-DATA.
   02  FILLER PICTURE X(59).
   02  FIELD-A PICTURE 9(8)V99 COMPUTATIONAL-3.
   02  FILLER PICTURE X(123).
   02  FIELD-B PICTURE X(8).
FD  TRANSACTION-FILE
   RECORDING MODE IS F
   RECORD CONTAINS 80 CHARACTERS
   LABEL RECORDS ARE OMITTED
   DATA RECORD IS TRANSACTION.
01  TRANSACTION.
   02  SORT-INDEX PICTURE X(8).
   02  DIGIT PICTURE 9.
      88 DELETION VALUE IS 1.
      88 QUOTIENT VALUE IS 2.
   02  DIVISOR PICTURE 9(6)V99 COMPUTATIONAL-3.
   02  FILLER PICTURE X(63).
FD  PRINT-FILE
   RECORDING MODE IS F
   RECORD CONTAINS 80 CHARACTERS
   LABEL RECORDS ARE OMITTED
   DATA RECORD IS PRINT-RECORD.
01.  PRINT-RECORD PICTURE X(80).
PROCEDURE DIVISION.
   OPEN INPUT OLD-MASTER TRANSACTION-FILE OUTPUT NEW-MASTER
   PRINT-FILE.
PARA-1.  READ TRANSACTION-FILE RECORD AT END GO TO PARA-4.
PARA-2.  READ OLD-MASTER RECORD AT END GO TO PARA-5.
```

Source Program Statements

```
PARA-3.  IF FIELD-B OF OLD-MASTER-DATA IS LESS THAN SORT-INDEX
           WRITE NEW-MASTER-DATA FROM OLD-MASTER-DATA
           GO TO PARA-2.
           IF FIELD-B OF OLD-MASTER-DATA IS GREATER THAN SORT-INDEX
           DISPLAY 'ERROR' SORT-INDEX UPON CONSOLE.
READ TRANSACTION-FILE RECORD AT END GO TO PARA-3.
IF DELECTION GO TO PARA-1.
IF QUOTIENT DIVIDE DIVISOR INTO FIELD-A OF OLD-MASTER-DATA
WRITE NEW-MASTER-DATA FROM OLD-MASTER-DATA
           GO TO PARA-1.
           WRITE PRINT-RECORD FROM TRANSACTION AFTER
           ADVANCING 2 LINES.
           WRITE NEW-MASTER-DATA FROM OLD-MASTER-DATA.
           GO TO PARA-1.
           NOTE THAT THE FOLLOWING PARAGRAPHS ARE
           EXECUTED ONLY AFTER ALL TRANSACTIONS
           HAVE BEEN PROCESSED.
PARA-4.  READ OLD-MASTER-DATA AT END GO TO PARA-5.
           WRITE NEW-MASTER-DATA FROM OLD-MASTER-DATA.
           GO TO PARA-4.
PARA-5.  CLOSE OLD-MASTER TRANSACTION-FILE NEW-MASTER
PRINT-FILE.
           STOP RUN.
```

Discussion

Prior to the PROCEDURE DIVISION of the Solution Program, there are two items that bear discussion. The first is the use of the same name for fields in separate files (FIELD-A and FIELD-B in both the OLD-MASTER and NEW-MASTER files). The second is the use of level indicator 88 (in the TRANSACTION record).

Field names, when used in the PROCEDURE DIVISION, must not be ambiguous. Uniqueness may be achieved in either of two ways: by unique definition or by qualification. Up until this Solution Program, each field name used has been uniquely associated with a single field, and this unique definition has removed the possibility of an ambiguous interpretation. Here, however, FIELD-A and FIELD-B have not been uniquely defined and require qualification when used to avoid ambiguity.

Qualification of a field name may be used to achieve uniqueness when such a name exists within a hierarchy of names. The field name being qualified is followed by OF or IN, then the next higher name in the hierarchy. Refer to PARA-3 of the PROCEDURE DIVISION, for example. FIELD-B is qualified by OF OLD-MASTER-DATA, making unambiguous which FIELD-B is meant. Qualification when not needed for uniqueness is permitted; this capability enables the language of COBOL to be more nearly like that of English. Also, several levels of qualification may be used where appropriate. The field defined as CONTACT in the following data description, for example,

```
01    ASSEMBLY
02    CHASSIS
03    POWER-SUPPLY
04    CONTACT PICTURE - - - - -
04    DIODE PICTURE - - - - -
04    TRANSISTOR PICTURE - - - -
```

could be used in qualified form as CONTACT OF POWER-SUPPLY IN CHASSIS OF ASSEMBLY in a program for processing a bill of materials. Note that even when qualification of a field name is possible, the same name must never be used for a field of data and for a procedure (section name or paragraph name in the PROCEDURE DIVISION).

Level indicators 01-49 are used to specify the hierarchical relationships among data fields in a file. Neither the special level indicator 77, used for defining fields not part of files, nor the special level indicator 88, encountered in the Solution Program, is involved in this hierarchical capability. Level 88, in particular, is a name assigned by the programmer to a value that may be assumed by the associated data item. Consider the usage in the Solution Program, reproduced here:

```
02  DIGIT PICTURE 9.
      88 DELETION VALUE IS 1.
      88 QUOTIENT VALUE IS 2.
```

DIGIT is the name of a field which consists of a single numeric digit. DELETION and QUOTIENT, however, are not field names. They are used as the names of DIGIT when the value of the latter is 1 or 2, respectively. Refer to the PROCEDURE DIVISION statement, IF DELETION GO TO Para-1. This statement is equivalent to: IF DIGIT IS EQUAL TO 1 GO TO PARA-1.

The logical tests written into the PROCEDURE DIVISION to enable the appropriate alternatives to be taken in the updating of the OLD-MASTER file are incorporated into a number of IF statements. None of these uses the ELSE (or OTHERWISE) option. When this option is omitted, control passes to the next sentence if the condition is false. Thus the statement, IF DELETION GO TO PARA-1, is equivalent to the statement, IF DELETION GO TO PARA-1 OTHERWISE NEXT SENTENCE.

PROBLEMS

1. What makes the PROCEDURE DIVISION of a COBOL program read like English (as compared to FORTRAN and Assembler Language)? Is this an advantage? Why?
2. The COBOL language was developed after the FORTRAN language had been developed. Why did not FORTRAN suffice? What can be done with COBOL that cannot be done with FORTRAN?
3. What is the purpose of the Margin A-Margin B rules in the COBOL language?

4. What is the advantage of blocking logical records into physical records?
5. What are the advantages of being able to define hierarchies of data in a programming language? Why is this capability not incorporated into every programming language?

A System/360 installation includes a card reader (2540), printer (1403), and two magnetic tape units (2400), and designates these CR, PR, T1, and T2, respectively. The processor is Model 50, and the memory size is E(32,768 bytes). A data file is punched into cards, one record per card, and sorted into ascending sequence according to Part Number in the following record format:

Columns 1-8		Location
1-2	State Code (99)	
3	Warehouse (9)	
4-5	Floor (99)	
6-8	Bin (999)	
Columns 9-36		Identification
9-15	Part Number (xxxxxxx)	
16-36	Part Name (x(21))	
37-38	Unit of Measure (xx)	
39-44	Unit Cost (999v999)	
Columns 45-80		Quantity
45-50	On Hand (999999)	
51-56	On Order (999999)	
57-62	Reorder Point (999999)	
63-68	Average Sales (999999)	
69-74	Sales This Month (999999)	
75-80	Sales Last Month (999999)	

6. Write the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, and DATA DIVISION of a program to place the given data file on magnetic tape, blocking 20 logical records in each physical record.
7. Write the PROCEDURE DIVISION of the above card-to-tape program, augmenting the other divisions as necessary. Check each record for correct sequence, displaying to the operator the Part Number of any records out of order. Also, display to the operator the Location and Identification of any records for which the sum of On Hand and On Order is less than the Reorder Point.
8. Write a program which uses the magnetic tape file created above to produce a printed report. For each record, this report contains a line showing Part Name, Part Number, Unit Cost, value of On Hand, value of On Order, and the total value of On Hand and On Order. (It is not necessary to use Unit of Measure in this computation.) In addition, this report contains a final total line showing the total value of On Hand for all records, the total value of On Order for all records, and the sum of these amounts.
9. Write a program to update the above magnetic tape file. Updating is in accordance with a set of transaction cards sorted in the same

sequence as the tape records and punched in the following format:

Columns 1-44	Same as Tape Records
Column 45	Transaction Code
1—Receipt	
2—Issue (Sale)	
3—Order	
Columns 46-51	Quantity

The updated file should be created according to the following rules:

1. When the Transaction Code is 1, Quantity is added to On Hand and subtracted from On Order.
2. When the Transaction Code is 2, Quantity is subtracted from On Hand and added to Sales This Month.
3. When the Transaction Code is 3, Quantity is added to On Order.

Each tape record may have none, one, or several transaction records associated with it. Do not include error checking in this program.

10. Write a single program which combines the updating of Exercise 9, the report of Exercise 8, and the checking of Exercise 7.
11. Compare the use of READ and WRITE with the use of ACCEPT and DISPLAY, respectively.
12. Compare the use of READ and MOVE with the READ . . . INTO option; compare the use of MOVE and WRITE with the WRITE . . . FROM option.
13. What is the difference in the results obtained by the statements SUBTRACT A FROM B and SUBTRACT A FROM B GIVING C?
14. What is the difference between STOP RUN and STOP 'RUN'?
15. What are the computational conditions that can produce a SIZE ERROR?

Chapter 16

Data Description and File Organization

This chapter presents a continuation and extension of some of the COBOL facilities already mentioned previously. We shall concentrate on a more detailed examination of the features used to allocate storage and define externally stored data files.

16.1 THE PICTURE SPECIFICATION

The PICTURE clause is used to provide a detailed description of fields which are not subdivided into subordinate fields and to specify special report editing. Both types of usage are fully described below.

The two alternative forms for the PICTURE clause are

PICTURE IS *description*
PICTURE *description*

Five forms of coded description are available:

1. Alphabetic form. The letter A is used to provide the coded description for alphabetic fields, which may contain any combination of the letters

A through Z and the blank (space). Both of the following examples specify alphabetic fields of six characters:

```
PICTURE IS AAAAAA.
PICTURE IS A(6).
```

2. Alphameric form. The letter X is used to provide the coded description for alphameric fields, which may contain any combination of the characters in the EBCDIC set.
3. Numeric form. The characters 9, S, V, and P are used to provide the coded description for numeric fields, which may contain any combination of the numeric digits 0–9 and which may include an algebraic sign (+ or −). The 9 is used to denote the position of each numeric digit in the field, and the S is used to denote the position of the sign, if any. If the number is not to be treated as an integer, V denotes the position of the (assumed) decimal point. Sometimes the position of the (assumed) decimal point is neither within nor adjacent to the digits of the field, and a positioning code is desired; this is the purpose of the P. The following examples show the use of the above coding in several PICTURE clauses:

Numeric Field	PICTURE *Clause*
12345	PICTURE IS S99999[a]
−12345	PICTURE IS S99999
+12345	PICTURE IS S99999V
−12.345	PICTURE IS S99V999
12345 million	PICTURE IS S99999PPPPPPV[b]

[a]Note that the S is always required when USAGE IS COMPUTATIONAL. Otherwise, S is written if and only if the field is to contain a sign.
[b]There are only five digits in the numeric field; it would be incorrect, then, to specify the PICTURE as S99999999999V or as S9(11)V.

Numeric fields may contain up through 18 digits, whether USAGE IS DISPLAY (zoned-decimal), USAGE IS COMPUTATIONAL-3 (packed-decimal), or USAGE IS COMPUTATIONAL (fixed-point binary). When USAGE IS COMPUTATIONAL, fields occupy a halfword, word, or doubleword, depending on whether the number of decimal digits is 1–4, 5–9, or 10–18, respectively.

4. Floating-point form. The characters 9, +, −, V, . , and E are used to provide the coded description for floating-point fields. Such a description has five parts which must occur in the following sequence: (a) + or −, (b) 1 through 16 9's representing the fraction, with a leading, embedded, or trailing decimal point or V, (c) the letter E, (d) + or −, and (e) 99 representing the exponent. Use of the + indicates that an actual plus sign designates positive values, while use of the − indicates that a blank designates positive values; negative values are always

designated by a minus sign. Following are examples of several external floating-point fields and their associated PICTURE clause.

External Field	PICTURE *Clause*
12345.E−03	PICTURE IS −9(5)VE+99
−12345.E+03	PICTURE IS −9(5)VE+99
+12345.E−03	PICTURE IS +9(5).E+99
+12.345E+03	PICTURE IS +99.999E+99
12.345E03	PICTURE IS −99.999E−99

In internal format, floating-point fields occupy a word or doubleword depending on whether the number of decimal digits specified is 1–7 or 8–16, respectively. In either case, the magnitude of a floating-point quantity may not exceed .72E+76 as with FORTRAN and Assembler Language.

5. Report form. The characters 9, V, and P (which have the same meaning as for numeric fields) and also the eleven characters . Z * CR DB , 0 B $ + and − are used to provide the coded description for the editing of fields to be used in reports. Such editing is carried out by means of the movement of data to a field with an appropriate PICTURE and is similar to, but simpler than, the use of the EDIT instruction in Assembler Language.

   The character 9, as with the PICTURE of numeric fields, specifies the position of numeric digits; however, the characters Z and * may be used for this purpose also. The latter characters, which may appear in the same PICTURE with 9's but never to their right, are used for the suppression of all zeros prior to the first significant digit. The Z causes leading zeros to be replaced by blanks, and the * causes leading zeros to be replaced by asterisks (for the purpose of check protection).

   The placement for the three characters , (comma), 0, and B in the coded description of the report field specifies that the comma, zero, and blank, respectively, are to appear in the corresponding positions of the report field itself. When they are placed among leading zeros whose suppression has been specified by either Z's or *'s, however, the commas, zeros, or blanks, as the case might be, are also suppressed.

   The character . (or, alternatively, the V) specifies that an actual decimal point is to be placed in the indicated position. The characters CR (credit symbol) and DB (debit symbol) may be used only at the right end of a PICTURE and designate that the specified symbol is to be placed in the indicated positions if the value of the field is negative; otherwise, these two positions are filled with blanks.

   The characters $, +, and − have two types of usage: single or "drifting" string. Used singly, the $ specifies that a dollar sign is to be inserted at the indicated position; the + or −, which must appear as either the first or last character in the PICTURE, specifies that a plus

TABLE 16.1 EDITING EXAMPLES

Prior to Editing		After Editing	
PICTURE	Data	PICTURE	Edited Data
s99999	12345	−zz,zz9.99	12,345.00
s9(5)	00100	Szz,zz9.99	S 100.00
s9(5)v	00000	−zz,zz9.99	0.00
9(5)	00000	$zz,zzz.zz	
999v99	12345	$zz,zz9.99	S 123.45
9(5)	00123	$**,**9.99	$***123.00
9(5)	00123	$$$,$$9.99	$123.00
9(4)v9	12345	$$$,$$9.99	$1,234.50
s9(5)	−12345	zzzz9.99−	12345.00−
s9(5)	00123	------99	123.00
s9(5)	−00001	------.99	−1.00
s9(5)	12345	+zzzzz.99	+12345.00
s9(5)	−12345	+zzzzz.99	−12345.00
9(5)	12345	bb999.00	345.00
s9(5)	−12345	zzzzz.99CR	12345.00CR
s9(5)	12345	$$$$$.99CR	$12345.00

sign, minus sign, or blank is to be inserted in the indicated position, according to the same interpretation for + and − as given above for the floating-point form. Used as a drifting string, that is, a series of leading characters either continuous or interrupted only by the , 0, B, or v characters, a S or + or − is inserted immediately to the left of the first nonzero digit, and blanks are inserted in all positions to the left.

A set of examples of the editing capability of COBOL is given in Table 16.1. In each example, the PICTURE and data are given both prior to editing and after editing.

Additional rules for constructing the PICTURE of a report field are:

1. No more than one drifting string may be used.
2. At least one character must refer to a digit position.
3. If there are no 9's in a PICTURE specification, BLANK WHEN ZERO is implied unless all numeric positions contain asterisks. BLANK WHEN ZERO is a clause that may be used in connection with the description of report fields. It specifies that the field, regardless of the PICTURE, is to be filled with blanks whenever its value is zero.
4. The use of one form of sign control precludes the use of other forms.
5. z may not follow * or 9 or a drifting string.
6. * may not follow 9 or z or a drifting string.
7. Only one type of digit position character may appear to the right of the decimal point.

8. If one of the numeric character positions to the right of the decimal point is represented by +, −, $, z, or *, then all the numeric character positions in the PICTURE must be represented by the same character.
9. A floating string must begin with at least two consecutive appearances of the +, −, or $.

16.2 FILE PROCESSING

In writing programs to create and/or process files, both the organization of such files and the type of access to them must be considered. By *organization* is meant the manner in which the component records are positioned in a file and in which control fields are used to record such positions. By type of *access* is meant the manner by which the order of reading and writing records is controlled. System/360 COBOL provides four types of file organization capabilities and two types of access methods (but only seven of the eight possible organization/access combinations); yet the programming examples and exercises so far encountered have almost completely ignored both organization and access. The following discussion will illuminate the material of the previous chapter and also provide background for this one.

16.2.1 File Definition

Files stored on media such as magnetic tape or punched cards are constrained by the nature of these media and of the devices which transport them, so that both the organization and access to these records is sequential. While sequential organization and access may also be applied to files stored on direct-access devices such as disk units (in fact, the Solution Program of Example 81 in the previous chapter could have been oriented to the 2311 rather than to the 2400 by changing only the ASSIGN clauses), there are no alternatives for devices specified either as UNIT-RECORD or as UTILITY in ASSIGN clauses.

Files whose ASSIGN clause designates usage of a DIRECT-ACCESS unit are not subject to the above constraint. They may be organized sequentially and in three other ways (described below): indexed, direct, and relative. They may be accessed in sequential order and also in arbitrary (random) order. Both organization and type of access are specified by clauses (or the lack of such clauses) in the SELECT sentence. The various alternatives and the clauses associated with each are given in Table 16.2. Note that the clause ACCESS IS SEQUENTIAL is equivalent to no access clause at all.

The records of a file with sequential organization are physically stored in the order in which they are encountered, with no consideration given to their contents. An indexed file's records are also physically stored in the

TABLE 16.2 ASSIGNMENT OF FILE ATTRIBUTES

Organization	Sequential Access	Random Access
Sequential	No ORGANIZATION clause ACCESS IS SEQUENTIAL	—
Indexed	ORGANIZATION IS INDEXED, ACCESS IS SEQUENTIAL	ORGANIZATION IS INDEXED, ACCESS IS RANDOM
Direct	ORGANIZATION IS DIRECT, ACCESS IS SEQUENTIAL	ORGANIZATION IS DIRECT, ACCESS IS RANDOM
Relative	ORGANIZATION IS RELATIVE, ACCESS IS SEQUENTIAL	ORGANIZATION IS RELATIVE, ACCESS IS RANDOM

order in which they are encountered; however, they must be presorted according to a field designated as the RECORD KEY in the SELECT sentence. When the file is created, the contents of the RECORD KEY fields are used to set up an index, enabling random as well as sequential access to each record. When the file is processed, the index structure is used to relate RECORD KEY fields to the appropriate physical addresses.

Neither the records of files with *direct organization* nor those with *relative* organization are ordinarily stored in the physical order in which they are encountered. The storage location of each record is individually specified

TABLE 16.3 2311 DISK STORAGE PROPERTIES FOR FIXED-LENGTH RECORDS (PER TRACK)

Number of Records Per Track	Length	Track Capacity (Without Key)	Length	Track Capacity (With Key)
1	3625	3625	3605	3605
2	1739	3478	1719	3438
3	1130	3390	1110	3330
4	829	3316	809	3236
5	650	3250	630	3150
6	531	3186	511	3066
7	446	3122	426	2982
8	383	3064	363	2904
9	334	3006	314	2826
10	294	2940	274	2740
11	262	2882	242	2662
12	235	2820	215	2580
13	212	2756	192	2496
14	193	2702	173	2422
15	176	2640	156	2340
16	161	2576	141	2256
17	148	2516	128	2176
18	137	2466	117	2106
19	127	2413	107	2033
20	117	2340	97	1940

and may be arbitrarily assigned or may be a function of one or more fields in the records themselves. Whichever is the case, the positions of records in files with direct organization are specified by means of an ACTUAL KEY field, which designates the track relative to the first track for the file, and a SYMBOLIC KEY field, which identifies the record on the track.

The positions of records in files with relative organization are specified by means of a SYMBOLIC KEY field alone; here, the contents of this field designate the position of the record relative to the first record in the file.

An important point to note is that the various types of file organization are meaningful only in their context and are mutually incompatible with each other. Thus within COBOL a file can be processed only with the same organization with which it was created.

16.2.2 Files for the 2311 Disk Unit

As far as the COBOL programmer is concerned, the disk pack for the 2311 unit consists of 2,000 tracks, each track providing space for at least one record. The number of available bytes depends on blocking factor and type of file organization. Table 16.3 summarizes these storage characteristics for records with and without identifying keys.

EXAMPLE 82

Write a program that uses a file of punched cards to create four files stored in a 2311 unit, one file each with sequential, indexed, direct, and relative organization. The card file (to be called C-FILE) consists of 50 records, each of 240 characters, and each punched onto three cards. These records are presorted into ascending sequence according to a three-digit field in positions 238–240.

Store the records ten per track. Assume the DD control statement has been used to allocate tracks 1–5 to the sequential file (called S-FILE), 6–10 to the indexed file (called I-FILE), 11–15 to the direct file (called D-FILE), and 16–20 to the relative file (which we'll call R-FILE).

Solution Procedure

Both the sequential and indexed files are produced with the records in the same order as those in the card file, but in the latter case the card file must be sorted into ascending order according to a field designated as the RECORD KEY (named IR-KEY). If the IR-KEY values are not, in fact, in ascending order, the indexed file cannot be produced.

Both the direct and relative files may have their records placed in any order. In the former case the location is designated by the ACTUAL KEY (which we shall name DA-KEY) and SYMBOLIC KEY (which we will call DRS-KEY), and in the latter case, the location being designated just by the SYMBOLIC KEY (DRS-KEY). For simplicity and because no other way is meaningful in this context, the records are placed in the same order as those in the card file.

Solution Program

Source Program Statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'FILES'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360 E50.
OBJECT-COMPUTER. IBM-360 E50.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT C-FILE ASSIGN TO F001 UNIT-RECORD 2540 UNIT
  RESERVE 1 ALTERNATE AREA.
  SELECT S-FILE ASSIGN TO F015 DIRECT-ACCESS 2311 UNIT.
  SELECT I-FILE ASSIGN TO F015 DIRECT-ACCESS 2311 UNIT
  ORGANIZATION IS INDEXED RECORD KEY IS IR-KEY.
  SELECT D-FILE ASSIGN TO F015 DIRECT-ACCESS 2311 UNIT
  ORGANIZATION IS DIRECT ACTUAL KEY IS DA-KEY
  SYMBOLIC KEY IS DRS-KEY.
  SELECT R-FILE ASSIGN TO F015 DIRECT-ACCESS 2311 UNIT
  ORGANIZATION IS RELATIVE SYMBOLIC KEY IS DRS-KEY.
DATA DIVISION.
FILE SECTION.
FD C-FILE RECORDING MODE IS F
  RECORDS CONTAIN 240 CHARACTERS
  LABEL RECORD IS OMITTED
  DATA RECORD IS C-RECORD.
01  C-RECORD.
  02  FILLER PICTURE X(237).
  02  FIELD PICTURE 999 COMPUTATIONAL-3.
FD S-FILE RECORDING MODE IS F
  BLOCK CONTAINS 10 RECORDS
  RECORD CONTAINS 240 CHARACTERS
  LABEL RECORD IS STANDARD
  DATA RECORD IS S-RECORD.
01  S-RECORD PICTURE X(240).
FD I-FILE RECORDING MODE IS F
  BLOCK CONTAINS 10 RECORDS
  RECORD CONTAINS 240 CHARACTERS
  LABEL RECORD IS STANDARD
  DATA RECORD IS I-RECORD
01  I-RECORD PICTURE X(240).
FD D-FILE RECORDING MODE IS F
  RECORD CONTAINS 240 CHARACTERS
  LABEL RECORD IS STANDARD
  DATA RECORD IS D-RECORD.
01  D-RECORD PICTURE X(240).
```

Source Program Statements

```
FD R-FILE RECORDING MODE IS F
  RECORD CONTAINS 240 CHARACTERS
  LABEL RECORD IS STANDARD
  DATA RECORD IS R-RECORD.
01  R-RECORD PICTURE X(240).
WORKING-STORAGE SECTION.
77  COUNTER-1 PICTURE S99 COMPUTATIONAL VALUE IS ZERO.
77  COUNTER-2 PICTURE S99 COMPUTATIONAL VALUE IS ZERO.
77  DA-KEY PICTURE S9(5) COMPUTATIONAL.
77  DRS-KEY PICTURE S9(8) COMPUTATIONAL.
PROCEDURE DIVISION.
OPENER. OPEN INPUT C-FILE OUTPUT S-FILE I-FILE D-FILE R-FILE.
PARA-1. ADD 1 TO COUNTER-1.
PARA-2. ADD 1 TO COUNTER-2.
  READ C-FILE AT END GO TO PARA-4.
  WRITE S-RECORD FROM C-RECORD.
  WRITE I-RECORD FROM C-RECORD INVALID KEY GO TO PARA-3.
  MOVE COUNTER-1 TO DA-KEY.
  MOVE COUNTER-2 TO DRS-KEY.
  WRITE D-RECORD FROM C-RECORD.
  COMPUTE DRS-KEY = 10 * COUNTER-1 + COUNTER-2 - 10.
  WRITE R-RECORD FROM CARD-RECORD.
TESTER. IF COUNTER-2 = 10 SUBSTRACT 10 FROM COUNTER-2
  GO TO PARA-1 OTHERWISE GO TO PARA-2.
PARA-3. DISPLAY 'CARDS OUT OF ORDER'.
PARA-4. CLOSE C-FILE S-FILE I-FILE D-FILE R-FILE.
  STOP RUN.
```

Although this example is not at all realistic, the creation of files with all four types of organization within a single program is instructive. First, consider the SELECT sentences in the FILE-CONTROL paragraph. The 2311 is designated as DIRECT-ACCESS rather than UTILITY. UTILITY restricts organization to sequential whereas DIRECT-ACCESS allows any organization. In each case the ACCESS IS SEQUENTIAL (because the clause is omitted), and the ORGANIZATION clauses are straightforward. The major differences come in the key clauses. Since a sequential file has no key clause, and no address is associated with the individual records, it cannot be accessed at random, even if it is created on a disk.

The indexed file uses the clause RECORD KEY IS IR-KEY to specify that the addressing key for each logical record is given by the contents of a field named IR-KEY, which is defined as a three-digit integer with COMPUTATIONAL-3 usage. In general, a field specified by a RECORD KEY may be any fixed-length field less than 256 bytes in length.

The direct file uses two keys to specify the address of each logical record. ACTUAL KEY IS DA-KEY gives the name of the field containing the actual track number of each record, this number being relative to the first track designated for the file.

Such a key is always defined in the way DA-KEY is defined in the Solution Program, in other words, COMPUTATIONAL with PICTURE S9(8), and may not exceed 65,535 in value. SYMBOLIC KEY IS DRS-KEY gives the name of the field containing the number of a record on a track. Here this key is defined as COMPUTATIONAL with PICTURE S9(8); in general utilization with respect to a direct file, such a key may be any fixed-length field whose length is less than 256 bytes.

Finally, the relative file also uses the clause, SYMBOLIC KEY IS DRS-KEY. Here the meaning differs from that associated with a direct file. DRS-KEY gives the name of the field containing the record number relative to the first record in the file. When used for this type of file, this key must be defined in the way DRS-KEY is defined, COMPUTATIONAL with PICTURE S9(8), and must be less than 16,777,216 in value.

Logical records may or may not be blocked in sequential and indexed files, but they cannot be blocked in direct and relative files, at least not by use of the BLOCK CONTAINS clause. If blocking is desired for the latter types of organization, the RECORD CONTAINS clause must specify the number of characters in the block, and the problem programmer assumes responsibility for record-by-record processing within each block. Second, the LABEL RECORD clause must specify OMITTED for files assigned to unit-record devices and STANDARD for files with indexed, direct, or relative organization: only when the file organization is sequential and when the file is assigned to a utility or direct-access device may the programmer choose to omit labels, to use standard labels only, or to use both standard and user labels.

The INVALID KEY GO TO PARA-3, contained in the second WRITE statement, provides the problem programmer with a convenient means of specifying what action to take should the fields referenced by the RECORD KEY clause be out of sequence. (Recall that in a file with indexed organization, the records must be sorted into ascending order, the basis for such ordering being the field referenced by the RECORD KEY clause). The words INVALID KEY are followed by whatever statement is appropriate to the situation; the GO TO used here is the statement most often found in this context.

In format, the WRITE statements which transfer records for both the direct and the relative files are identical to that associated with the sequential file, but this is somewhat misleading. The location to which the referenced records are transferred depends not on where the previous record was written, as it does with the sequential file, but on the contents of the keys. In the Solution Program, the two MOVE statements set the values of the fields designated as the ACTUAL KEY and the SYMBOLIC KEY to their desired values for the direct file, while the COMPUTE statement sets the value of the field designated as the SYMBOLIC KEY to its desired value for the relative file.

16.3
TESTING FOR END OF PAGE

The ENVIRONMENT DIVISION may include an I-O-CONTROL paragraph which can be used to specify certain COBOL-directed actions. We shall be concerned at present with a form of the APPLY clause which allows us to check for the end of a page of printed output. This form is

```
I-O CONTROL.
    APPLY v1 TO FORM-OVERFLOW ON filename
```

where *filename* is the name assigned to the print file and *v*1 is a programmer-defined variable whose PICTURE IS 9. The keyword FORM-OVERFLOW causes COBOL to use *v*1 for storing the result of an end of page test wherever such a test is called for in the PROCEDURE DIVISION. Correspondingly, the PROCEDURE DIVISION would include a test symbolized as

```
IF v1 action1 ELSE action2
```

where *action*1 and *action*2 may be GO TO's or other statements. When the end of a print page is encountered, COBOL automatically places a 1 in *v*1, so that *action*1 is taken when the IF is processed. Otherwise, a 0 is maintained in *v*1. This allows the programmer to use a WRITE statement with the ADVANCING option, giving him the facility to include special page headings, etc.

16.4 THE TRANSFORM STATEMENT

Character substitution can be handled by a statement whose general form is

```
TRANSFORM v1 CHARACTERS FROM code1 TO code2
```

*v*1 is a variable in alphabetic, alphameric or report form. The keyword CHARACTERS is optional here and the items in *code*1 and *code*2 may be figurative constants, nonnumeric literals or variable names. In general, individual characters specified by *code*1 are transformed according to the second code. The type of transformation is determined by one of the nine combinations of codes as outlined below:

1. When both codes are figurative constants, all characters in *v*1 that match *code*1 are replaced by *code*2. For example, the statement

   ```
   TRANSFORM X FROM ZEROS TO SPACES
   ```

   will replace all zeros in X with blanks. In addition to ZEROS and SPACES, the figurative constants ZERO, ZEROS, SPACE, QUOTE, QUOTES, HIGH-VALUE, HIGH-VALUES, LOW-VALUE, and LOW-VALUES may be used.
2. When *code*1 is a figurative constant and *code*2 is a one-character nonnumeric literal, all characters in *v*1 that match *code*1 are replaced by *code*2. Thus,

   ```
   TRANSFORM X FROM ZEROS TO ' '
   ```

   will do the same thing as the example in the previous paragraph.

3. When *code*1 is a figurative constant and *code*2 is the name of a one-character variable, all of *v*1's characters that match *code*1 are replaced by *code*2's contents. Thus, if Y contains the single character $, then

```
TRANSFORM X FROM SPACES TO Y
```

   replaces all blanks in X with $'s.
4. When *code*1 is a nonnumeric literal and *code*2 is a figurative constant, all of *v*1's characters which match *code*1 are replaced by *code*2. Thus,

```
TRANSFORM X FROM 'ABC' TO SPACES
```

   will replace all A's, B's, and C's in X with blanks.
5. When both *code*1 and *code*2 are nonnumeric literals and of the same length, each *code*1 character is replaced by the corresponding one in *code*2's sequence. Otherwise, *code*2 must be a single character, in which case any of the characters listed in *code*1 is replaced by the one in *code*2. For example, the following two statements

```
TRANSFORM X FROM 'AB' TO 'BA'
TRANSFORM Y FROM 'ABC' TO 'W'
```

   cause all A's and B's in X to be replaced by B's and A's, respectively, and all A's, B's, and C's in Y to be replaced by W's.
6. The rules stated in the previous paragraph apply when *code*1 is a nonnumeric literal and *code*2 is a variable name. Thus, if X contains WBG and Y contains the single character T, the statements

```
TRANSFORM Z FROM 'ABC' TO X
TRANSFORM Z FROM 'DJK' TO Y
```

   will replace all A's, H's, and C's in Z with W's, B's, and G's, respectively. Furthermore, all D's, J's, and K's in Z will be replaced by T's.
7. When *code*1 is a variable name and *code*2 is a figurative constant, all characters in *v*1 which match characters in *code*1 will be replaced by *code*2. For instance, the statement

```
TRANSFORM X FROM Y TO ZEROS
```

   will replace with zeros all characters in X which also appear in Y.
8. When *code*1 is a variable name and *code*2 is a nonnumeric literal, the rules and constraints stated in paragraph 5 apply. Thus, if Y contains 1234, and Z contains 5678, the statements

```
TRANSFORM X FROM Y TO 'ABCD'
TRANSFORM X FROM Z TO '$'
```

   will change all 1's, 2's, 3's, and 4's in X to A's, B's, C's, and D's, respectively, and will replace all 5's, 6's, 7's, and 8's in X with $'s.

9. When *code*1 and *code*2 are both variables, they must be of equal length. Any time a character in *v*1 matches one of those in *code*1, it is replaced by the one in the same position of *code*2.

EXAMPLE 83

A sequential file resident on a 2311 consists of 176-character records blocked 15 per track. From this file, a printed report is to be prepared. The report form is 72 positions wide and is to contain one heading line per page and one body line per 2311 record. Write a program to produce this report according to the following specifications:

1. Each heading line is to have the title, SPECIAL REPORT, and the page number, as PAGE NO. XXX.
2. Each body line is to have three fields separated by blanks. Positions 1-10 are to be derived from positions 100-109 of the 2311 record; positions 21-50 from positions 31-60 of the 2311 record; and positions 61-72 from positions 61-64 of the 2311 record.
3. Positions (100-109, 31-60), and 61-64 of the 2311 record are alphameric and packed-decimal fields, respectively. Positions 100-109 are to be edited such that each A is printed as a B, each X or Y is printed as a Z, and each Z is printed as a 1. Positions 31-60 are to be printed without editing. Positions 61-64 are to be edited in such a way that the field, which may be positive or negative, is printed as a dollars-and-cents figure.

Solution Program

Source Program Statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'REPORT'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360 E50.
OBJECT-COMPUTER. IBM-360 E50.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT DISK-FILE ASSIGN TO F015 DIRECT-ACCESS 2311 UNIT
  RESERVE 1 ALTERNATE AREA.
  SELECT REPORT-FILE ASSIGN TO F003 UNIT-RECORD 1403 UNIT.
I-O-CONTROL.
  APPLY END-PAGE TO FORM-OVERFLOW ON REPORT-FILE.
DATA DIVISION.
FILE SECTION.
FD DISK-FILE RECORDING MODE IS F
  BLOCK CONTAINS 15 RECORDS
  RECORD CONTAINS 176 CHARACTERS
  LABEL RECORD IS STANDARD
  DATA RECORD IS DISK-RECORD.
```

Source Program Statements (continued)

```
01  DISK-RECORD.
    02  FILLER PICTURE X(30).
    02  FIELD-1 PICTURE X(30).
    02  FIELD-2 PICTURE S9(7) COMPUTATIONAL-3.
    02  FILLER PICTURE X(35).
    02  FIELD-3 PICTURE X(10).
    02  FILLER PICTURE X(67).
FD  REPORT-FILE RECORDING MODE IS F
    RECORD CONTAINS 72 CHARACTERS
    LABEL RECORD IS OMITTED
    DATA RECORD IS REPORT-RECORD.
01  REPORT-RECORD.
    02  FIELD-4 PICTURE X(10).
    02  FILLER PICTURE X(10).
    02  FIELD-5 PICTURE X(30).
    02  FILLER PICTURE X(10).
    02  FIELD-6 PICTURE $$$$$$$.99CR BLANK WHEN ZERO.
WORKING-STORAGE SECTION.
77  BLANK-LINE PICTURE IS X(72) VALUE IS SPACES.
77  UNITY PICTURE IS 999 VALUE IS 1.
77  END-PAGE PICTURE IS 9.
01  HEADING-LINE.
    02  FILLER PICTURE IS X(15) VALUE IS 'SPECIAL REPORT'.
    02  FILLER PICTURE IS X(49) VALUE IS SPACES.
    02  FILLER PICTURE IS X(5) VALUE IS 'PAGE'.
    02  PAGE PICTURE IS 999 VALUE IS 1.
PROCEDURE DIVISION.
PARA-1. OPEN INPUT DISK-FILE OUTPUT REPORT-FILE.
PARA-2. MOVE HEADING-LINE TO REPORT-RECORD.
    WRITE REPORT-RECORD AFTER ADVANCING 0 LINES.
    MOVE BLANK-LINE TO REPORT-RECORD.
    WRITE REPORT-RECORD AFTER ADVANCING 2 LINES.
PARA-3. READ DISK-FILE AT END GO TO PARA-4.
    TRANSFORM FIELD-3 CHARACTERS FROM 'AXYZ' TO 'BZZ1'.
    MOVE FIELD-3 TO FIELD-4.
    MOVE FIELD-1 TO FIELD-5.
    MOVE FIELD-2 TO FIELD-6.
    WRITE REPORT-RECORD AFTER ADVANCING 1 LINES.
    IF END-PAGE ADD UNITY TO PAGE GO TO PARA-2.
    ELSE GO TO PARA-3.
PARA-4. CLOSE DISK-FILE REPORT-FILE.
    STOP RUN.
```

Note the PICTURE and USAGE of FIELD-2. Because FIELD-2 is a packed-decimal field containing 7 digits, its length must then be 4 bytes (characters).

Note also the PICTURE of FIELD-6. When the packed-decimal FIELD-2 is moved to FIELD-6 in the PROCEDURE DIVISION, this PICTURE edits the quantity into a dollar sign preceding the first nonzero digit, followed by the digits with a decimal point inserted in the appropriate position. When the quantity is negative, the CR is printed to the right, and further, the BLANK WHEN ZERO clause produces spaces only when the quantity is zero.

This Solution program also illustrates that hierarchical level numbers, as well as the special level 77, may be used in the WORKING-STORAGE SECTION. When this is done, all the level 77 fields must de defined prior to definition in terms of the level numbers 01-49.

After opening the files in the PROCEDURE DIVISION, the Solution Program sets up the HEADING-LINE, which has been preset to include PAGE 001, skips output form to the beginning of a new page, and prints the heading. Recall that AFTER ADVANCING 0 LINES causes the output form to be skipped to the first printing line of the next page. Two blank lines are then "printed" by writing a blank record AFTER ADVANCING 2 LINES. This provides for increased readability of the printed report by separating the heading line from the body lines.

The next set of statements reads a 2311 record, creates a body line from it, and prints the body line on the report. The IF statement sends control back to PARA-3 to read the next disk record if the end-of-page has not yet been sensed. When it is sensed, PAGE is incremented by 1 to set up HEADING-LINE for the next page and the above procedure is repeated. The procedure ends when the end of the 2311 file is reached.

16.5
DATA AGGREGATES

16.5.1
Specification of Data Aggregates

When a number of data items share a set of common properties and are to be referred to collectively, COBOL provides facilities for doing this by means of the OCCURS clause. This is used as part of a data item description in accordance with the following general form:

*level v*1 PICTURE *description* OCCURS *n* TIMES

where *level* may not be 01 or 88, description is any legitimate PICTURE specification and *n* is an integer indicating the number of items in the data collection named *v*1. Each of the *n* members of such a collection is then referred to by the collective name supplemented by a parenthesized subscript. For example, if we specify

02 A PICTURE X(4) OCCURS 8 TIMES

we define an array A consisting of 32 characters divided into eight fields

of four characters each. The first of these is referred to as A(1), the second is A(2) and so on. Variable names may also be used as subscripts.

Alternatively, it is possible to define a collection with a variable number of elements by writing

level v1 PICTURE *description* OCCURS *nmax* TIMES DEPENDING ON v2

this time *nmax*, still an integer, refers to the *maximum* number of elements in the collection v1. v2 is a variable which must be described as an integer and whose current value defines the number of elements in v1 at that time.

In either form the OCCURS clause may be used to define one-, two-, or three-dimensional arrays. For instance, the following example defines a rectangular matrix of two rows and three columns.

```
01  ARRAY.
02  VECTOR OCCURS 2 TIMES.
03  ELEMENT OCCURS 3 TIMES
        PICTURE S9(9) USAGE COMPUTATIONAL.
```

Elements of this matrix are referred to by using a subscript consisting of row number followed by the column number. Thus ELEMENT(1,3) is the element in the first row, third column.

16.5.2 Cyclic Operations with Data Aggregates

COBOL makes it possible to define a sequence of operations which is to be executed a specified number of times. Furthermore, this cyclic processing can be carried out using a different array element each time, so that an entire array can be processed automatically.

The PERFORM statement specifies a transfer of control from one portion of a program to another, in order to execute some procedure a specified number of times, or until a condition is satisfied. It directs that control is to be returned to the statement immediately following the point from which the transfer was made. The several forms of this statement are illustrated below:

1. PERFORM PROCEDURE-1
   All statements in PROCEDURE-1 are executed once. PROCEDURE-1, and the other procedure names in following illustrations, are the names of sections and/or paragraphs in the PROCEDURE DIVISION.
2. PERFORM PROCEDURE-1 THRU PROCEDURE-2
   All statements in the contiguous set of procedures from PROCEDURE-1 THRU PROCEDURE-2 are executed once.
3. PERFORM PROCEDURE-1 6 TIMES
   The number of iterations is specified by a positive integer whose value is less than 32,768. This integer may be a literal, as shown here, or may be the contents of a field named in the PERFORM statement.

4. PERFORM PROCEDURE-1 THRU PROCEDURE-2 *v*1 TIMES
   Here *v*1 is a variable which contains a positive integer less than 32,768. This is the number of iterations of the set of statements from PROCEDURE-1 THRU PROCEDURE-2.
5. PERFORM PROCEDURE-2 UNTIL *v*1 = *n*.
   The procedure specified is performed until the condition *v*1 = *n*, is met. All test conditions described in connection with the IF statement may be used here.
6. PERFORM PROCEDURE-1 THRU PROCEDURE-2 UNTIL *v*1 IS ZERO
   This is the extension of the previous example to a set of contiguous procedures. Note that a sign test is used here, whereas a relation test was used in the previous example. As with IF statements, the test conditions specified may be simple or compound.
7. PERFORM PROCEDURE-1
   VARYING *v*1 FROM *n*1
   BY *n*2 UNTIL *v*1 = *n*3
   In this example, PROCEDURE-1 is first executed with *v*1 = *n*1, then with *v*1 = *n*1 + *n*2, etc. until *n*1 = *n*3. Both the FROM and BY items may be field names as well as literals; UNTIL, as previously, may specify any test condition.
8. PERFORM PROCEDURE-1 THRU PROCEDURE-2
   VARYING *v*1 FROM *n*1
   BY *n*2 UNTIL *v*2 > *n*3
   This example extends the statements executed to a set of contiguous procedures, illustrates the use of a variable name with BY, and shows that the test condition given does not have to include the variable incremented with each iteration. Both of the last two examples may be augmented further by adding one or two groups of the VARYING-FROM-BY-UNTIL items. This produces nested loops with the order being from outermost to innermost.

EXAMPLE 84

An indexed file on a 2311 consists of 156-character records blocked 15 per track. Each record consists of six 25-character alphameric fields followed by a 6-character alphameric RECORD KEY. This file is to be updated via a card file in which each pair of cards contains one record. Each 160-character card record consists of numeric digits in positions 1–2, blanks in positions 3–4, and the same format as the 2311 records in positions 5–160. Write a program to carry out the updating according to the following considerations:

1. When column 1 of the card record contains a 1, the corresponding 2311 record (the one with the same key) is to be deleted.
2. When column 1 of the card record contains a 2, the corresponding 2311 record, in its entirety, is to be replaced by the card record (columns 5–160).
3. When column 1 of the card record contains a 3, only that field of the 2311

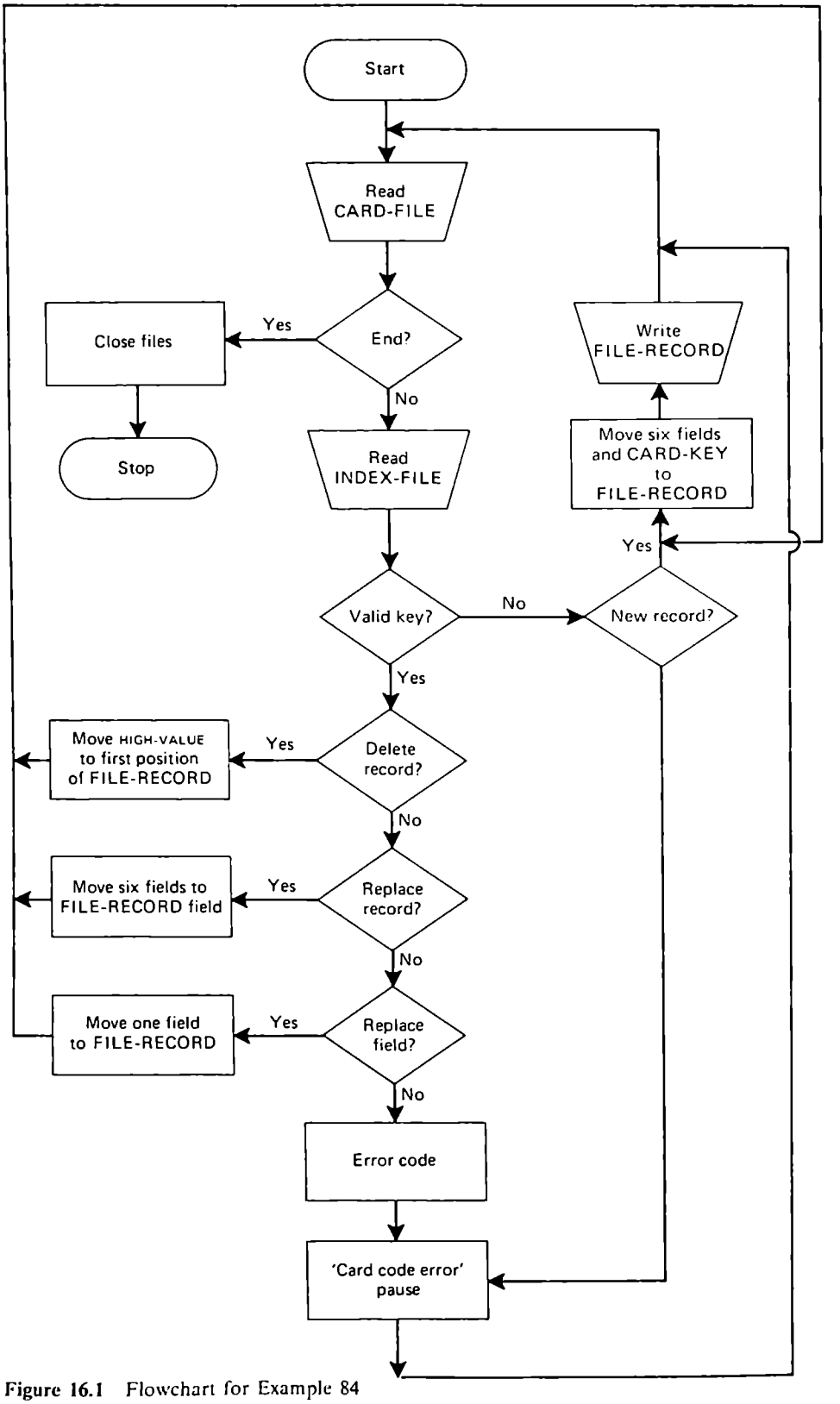


Figure 16.1 Flowchart for Example 84

record designated by the digit in column 2 is to be replaced by columns 5–29 of the card record.

4. When column 1 of the card record contains a 4, and when the card record key does not match the key of any 2311 record, a new record (columns 5–160 of the card record) is to be added to the 2311 file.

Solution Procedure

Deletion of a record in an indexed file is not a physical deletion. Instead, replacement of the first character with the figurative constant HIGH-VALUE causes the record to be treated as if it no longer exists.

Addition of a record to an indexed file, which is a file stored in ascending order of the records' keys, necessitates its insertion in a specific position between the record whose keys are lower and higher. Space is ordinarily not available to accomplish this physically. Instead, an overflow record scheme, which allows the programmer to treat all records as if they are in sequence physically, is used to produce the desired effect. The flowchart is shown in Figure 16.1.

Solution Program

Source Program Statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'INDEX'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360 E50.
OBJECT-COMPUTER. IBM-360 E50.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT CARD-FILE ASSIGN TO F001 UNIT-RECORD 2540 UNIT
  RESERVE 1 ALTERNATE AREA.
  SELECT INDEX-FILE ASSIGN TO F015 DIRECT-ACCESS 2311 UNIT
  RESERVE 1 ALTERNATE AREA ORGANIZATION IS INDEXED
  ACCESS IS RANDOM SYMBOLIC KEY IS CARD-KEY
  RECORD KEY IS DISK-KEY TRACK-AREA IS THIS-MANY CHARACTERS.
DATA DIVISION.
FILE SECTION.
FD CARD-FILE RECORDING MODE IS F
  RECORDS CONTAIN 160 CHARACTERS
  LABEL RECORD IS OMITTED
  DATA RECORD IS CARD-RECORD.
01  CARD-RECORD.
  02  UPDATE-CODE PICTURE 9.
        88  DELETE VALUE IS 1.
        88  REPLACE VALUE IS 2.
        88  REPLACE-SELECTIVELY VALUE IS 3.
        88  ADDITIONAL VALUE IS 4.
  02  DIGIT PICTURE IS 9.
  02  FILLER PICTURE X(2).
```

Source Program Statements

```
  02  CARD-RECORD-FIELD PICTURE X(25) OCCURS 6 TIMES.
  02  CARD-KEY PICTURE S9(6) COMPUTATIONAL.
FD INDEX-FILE RECORDING MODE IS F
  BLOCK CONTAINS 15 RECORDS
  RECORD CONTAINS 156 CHARACTERS
  LABEL RECORD IS STANDARD
  DATA RECORD IS FILE-RECORD.
01  FILE-RECORD.
  02  FILE-RECORD-FIELD PICTURE X(25) OCCURS 6 TIMES.
  02  DISK-KEY PICTURE S9(6) COMPUTATIONAL.
WORKING-STORAGE SECTION.
77  THIS-MANY VALUE IS 2340.
77  I PICTURE IS 9.
PROCEDURE DIVISION.
PARA-1. OPEN I-O INDEX-FILE INPUT CARD-FILE.
PARA-2. READ CARD-FILE AT END GO TO PARA-3.
  READ INDEX-FILE INVALID KEY GO TO PARA-4.
  IF DELETE MOVE HIGH-VALUE TO FILE-RECORD-FIELD (I) GO TO
  PARA-5 ELSE IF REPLACE MOVE 1 TO I PERFORM PARA-6
  6 TIMES GO TO PARA-5 ELSE IF REPLACE-SELECTIVELY MOVE
  CARD-RECORD-FIELD (I) TO FILE-RECORD-FIELD (DIGIT) GO TO
  PARA-5 ELSE IF NOT ADDITIONAL STOP 'CARD CODE ERROR'
  GO TO PARA-2.
PARA-5. WRITE FILE-RECORD GO TO PARA-2.
PARA-3. CLOSE INDEX-FILE CARD-FILE STOP RUN.
PARA-4. IF ADDITIONAL MOVE 1 TO I PERFORM PARA-6 6 TIMES
  MOVE CARD-KEY TO DISK-KEY GO TO PARA-5 ELSE STOP
  'CARD CODE ERROR' GO TO PARA-2.
PARA-6. MOVE CARD-RECORD-FIELD (I) TO FILE-RECORD-FIELD (I)
  ADD 1 TO I.
```

In this example, a file whose ORGANIZATION IS INDEXED and to which ACCESS IS RANDOM is to be updated according to information in a deck of cards. Both a SYMBOLIC KEY and a RECORD KEY are specified in the SELECT sentence, even though only the RECORD KEY was used in creating the indexed file. This is because the SYMBOLIC KEY, which in the Solution Program is read from a card, is used as a search argument, and the RECORD KEY, which is part of each 2311 record, is used as a table entry. When the SYMBOLIC KEY matches the RECORD KEY, the record sought has been located. If no such match can be made, the INVALID KEY clause of the READ sentence (see PARA-2 of PROCEDURE DIVISION) is used to specify the appropriate action to take.

The SELECT sentence for the indexed file also contains the clause: TRACK-AREA IS THIS-MANY CHARACTERS, in which THIS-MANY is a variable name. This clause is used to specify the available area for data records on each 2311 track to facilitate the addition of records to random-access files with indexed organization. This clause is required when variable-length records are added to the file and desirable, from the point of view of efficiency, when fixed-length records are added. The field named

in the TRACK-AREA clause must be shown with an 01 or 77 level number in the WORKING-STORAGE SECTION of the DATA DIVISION.

16.6 SEARCHING DIRECT FILES

When we have a file whose ORGANIZATION IS DIRECT we can exploit that file's random access capabilities. To do this, both an ACTUAL KEY and a SYMBOLIC KEY are required, the former to locate a particular track and the latter to identify the proper record on the track. The SYMBOLIC KEY field may be any fixed-length field less than 256 bytes in length, but the ACTUAL KEY field must have PICTURE S9(5) and USAGE COMPUTATIONAL. In addition, the value of this latter field must not exceed 65,535.

Addresses of records in random-access files are frequently derived from part numbers or other identification fields, and it often happens that such derivation produces the same file address for more than one record. When a file with direct organization is created, an attempt to write a record in a location in which a record has already been written initiates a search for the first unused area. This search begins on the track specified in the ACTUAL KEY field but continues onto subsequent tracks if necessary. When these records are read, the same search is carried out to locate the specific record desired. The I-O-CONTROL paragraph includes an APPLY RESTRICTED SEARCH clause, whose form is

```
APPLY RESTRICTED SEARCH OF n TRACKS
```

to limit the number of tracks searched, both in reading and writing records. If the desired space or record is not located, control is transferred to the INVALID KEY clause.

EXAMPLE 85

A file with direct organization is resident on a 2311 and consists of 100-byte records. Each record consists of twelve 3-byte packed-decimal integers, a 4-byte packed-decimal field which is the arithmetic mean of the first 12 fields and which is correct to the nearest hundredth, various other fields, and a 2-character alphameric record number. The first 12 fields represent historical data from each of the past 12 months.

Write a program to update the moving average of these records according to current data from a file of cards. Each card record contains a 3-digit track number, and a 5-digit data item.

Solution Procedure

A moving average is an arithmetic mean of historical data. As each new datum is obtained, it is included in the data and the oldest datum is dropped out. Computation of a new mean is then performed on a set of data which has *moved* one time period forward; hence *moving average*. Several periods of historical data must

be retained in order to use this method, and the computation must be programmed so that the oldest datum is always the one dropped out, no matter how many periods of data are processed.

Solution Program

Source Program Statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'DIRECT'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360 E50.
OBJECT-COMPUTER. IBM-360 H50.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT CARD-FILE ASSIGN TO 'F001' UNIT-RECORD 2540 UNIT
   RESERVE 1 ALTERNATE AREA.
   SELECT DIRECT-FILE ASSIGN TO 'F015' DIRECT-ACCESS 2311 UNIT
RESERVE 1 ALTERNATE AREA ACCESS IS RANDOM
   ORGANIZATION IS DIRECT ACTUAL KEY IS TRACK-NUMBER
   SYMBOLIC KEY IS RECORD-NUMBER.
I-O-CONTROL.
   APPLY RESTRICTED SEARCH OF 10 TRACKS ON DIRECT-FILE.
DATA DIVISION.
FILE SECTION.
FD CARD-FILE RECORDING MODE IS F
   RECORDS CONTAIN 10 CHARACTERS
   LABEL RECORD IS OMITTED
   DATA RECORD IS CARD-RECORD.
01  CARD-RECORD.
   02  TN PICTURE IS 999.
   02  RN PICTURE IS 99.
   02  DATUM PICTURE 9(5) COMPUTATIONAL-3.
FD DIRECT-FILE RECORDING MODE IS F
   RECORDS CONTAIN 100 CHARACTERS
   LABEL RECORD IS STANDARD
   DATA RECORD IS FILE-RECORD.
01  FILE-RECORD.
   02  DATA-REC OCCURS 12 TIMES PICTURE 9(5)
   USAGE IS COMPUTATIONAL-3.
   02  AVERAGE PICTURE9(5)V99 COMPUTATIONAL-3.
   02  FILLER PICTURE X(58).
   02  RECNO PICTURE S99 COMPUTATIONAL.
WORKING-STORAGE SECTION.
77  TOTAL PICTURE 9(7) COMPUTATIONAL-3.
77  TRACK-NUMBER PICTURE S9(5) USAGE COMPUTATIONAL.
77  RECORD-NUMBER PICTURE S9(5) USAGE COMPUTATIONAL.
PROCEDURE DIVISION.
```

Source Program Statements

```
PARA-1. OPEN INPUT CARD-FILE I-O DIRECT-FILE.
PARA-2. READ CARD-FILE RECORD AT END GO TO PARA-3.
  MOVE TN TO TRACK-NUMBER.
  MOVE RN TO RECORD-NUMBER.
  READ DIRECT-FILE RECORD INVALID KEY GO TO PARA-4.
  MOVE 1 TO RN.
  MOVE 2 TO TN.
  PERFORM PARA-5 UNTIL TN = 12.
  MOVE DATUM TO DATA-REC (12).
  MOVE ZERO TO TOTAL.
  PERFORM PARA-6 UNTIL TN = 1.
  DIVIDE 12 INTO SUM GIVING AVERAGE ROUNDED.
  WRITE FILE-RECORD INVALID KEY GO TO PARA-4.
  GO TO PARA-2.
PARA-3. CLOSE CARD-FILE DIRECT-FILE STOP RUN.
PARA-4. DISPLAY CARD-RECORD GO TO PARA-2.
PARA-5. MOVE DATA-REC(TN) TO DATA-REC(RN).
  ADD 1 TO RN.
  ADD 1 TO TN.
PARA-6. ADD DATA-REC (TN) TO TOTAL.
  SUBTRACT 1 FROM TN
```

16.7 COBOL SUBPROGRAMS

Although the PERFORM statement was introduced as a vehicle for processing sequences of instructions a specified number of times (16.5.2), its basic property (i.e., returning control to the statement immediately following the PERFORM statement) makes it the basic vehicle for setting up and handling COBOL subprograms. We saw (Example 84) that by setting up a separate paragraph (PARA-6 in that case) and referring to it with a PERFORM statement is, in effect, equivalent to setting up the statements in that paragraph as an internal procedure, branching to it, and returning.

The PERFORM statement may also be used, in conjunction with other COBOL features, to construct separately compilable routines and include them in more complex programs. This capability is provided through the use of the ENTER, CALL, and ENTRY statements.

16.7.1 Construction of Subprograms

COBOL subprograms start with the two statements

```
ENTER LINKAGE.
ENTRY 'name' USING param1 param2 . . . .
```

and end with the statements

```
RETURN.
ENTER COBOL.
```

name in this case represents the name of the subprogram, in other words, the name that will be used to invoke it from the calling program. *param*1, *param*2, ... indicates a list of parameters used in the subprogram. These correspond in number and properties to those passed to the subprogram in the argument list of a CALL statement. As we shall see, the names of these parameters need not be the same as those used as arguments in the calling program. However, in order to establish the identity and properties of these parameters, the main program's DATA DIVISION must include a LINKAGE SECTION in which each parameter is named and described by an appropriate PICTURE specification.

16.7.2 Transfer of Control to Subprograms

Subprograms may be called any number of times from any point in a COBOL program. In each case the linkage mechanism requires the following sequence of statements in the calling program:

```
ENTER LINKAGE.
CALL 'name' USING arg1, arg2, . . . .
ENTER COBOL.
```

*arg*1, *arg*2, ... represent the list of arguments to be used in the subprogram. These must agree in number, order and format (but not necessarily in name) with those listed in the subprogram as parameters and described in the calling program's LINKAGE SECTION.

EXAMPLE 86

A file with relative organization is resident on a 2311 and consists of 80-byte records. Each record consists of two 3-byte packed-decimal integers, various other fields and an 8-digit (4-byte) binary key. Write a program to process these records according to control information from a file of cards. Each record in the card file contains various fields in columns 1–71, a numeric digit in column 72, and a key in columns 73–80. Processing depends on the digit in column 72, as follows:

1. If a 1, the card contents are to replace the 2311 record having the same key.
2. If a 2, the sum of the squares of the two 3-byte 2311 fields is to be placed in positions 7–12 of the same 2311 record.
3. If any other digit, the two 3-byte 2311 fields are to be interchanged.

Solution Procedure

In addition to using this example to illustrate the random processing of records in a file with relative organization, it illustrates the subprogram capability of

COBOL. Therefore, the second and third of the above three alternatives are to be programmed as subprograms.

Solution Program

Source Program Statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'RELATIVE'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360 E50.
OBJECT-COMPUTER. IBM-360 E50.
INPUT-OUTPUT SECTION.
FILE-CONTROL. SELECT CARD-FILE ASSIGN TO 'F001'
  UNIT-RECORD 2540 UNIT RESERVE 1 ALTERNATE AREA.
  SELECT RELATIVE-FILE ASSIGN TO 'F015' DIRECT-ACCESS
  2311 UNIT RESERVE 1 ALTERNATE AREA ACCESS IS
  RANDOM ORGANIZATION IS RELATIVE SYMBOLIC
  KEY IS KEY-FIELD.
DATA DIVISION.
FILE SECTION.
FD CARD-FILE RECORDING MODE IS F
  RECORDS CONTAIN 80 CHARACTERS
  LABEL RECORD IS OMITTED
  DATA RECORD IS CARD-RECORD.
01  CARD-RECORD.
  02  FILLER PICTURE X(71)
  02  DIGIT PICTURE S9 COMPUTATIONAL-3.
  02  KEY-FIELD PICTURE S9(8) COMPUTATIONAL.
FD RELATIVE-FILE RECORDING MODE IS F
  RECORDS CONTAIN 80 CHARACTERS
  LABEL RECORD IS STANDARD
  DATA RECORD IS FILE-RECORD.
01  FILE-RECORD.
  02  I1 PICTURE S9(5) COMPUTATIONAL-3.
  02  I2 PICTURE S9(5) COMPUTATIONAL-3.
  02  I3 PICTURE S9(11) COMPUTATIONAL-3.
  02  FILLER PICTURE X(64).
  02  FIELD-KEY PICTURE S9(8) COMPUTATIONAL.
WORKING-STORAGE SECTION.
77  TEMPORARY PICTURE S9(5) COMPUTATIONAL-3.
LINKAGE SECTION.
77  A PICTURE S9(5) COMPUTATIONAL-3.
77  B PICTURE S9(5) COMPUTATIONAL-3.
77  C PICTURE S9(11) COMPUTATIONAL-3.
77  D PICTURE S9(5) COMPUTATIONAL-3.
PROCEDURE DIVISION.
P1.  OPEN INPUT CARD-FILE I-O RELATIVE-FILE.
```

Source Program Statements

```
P2.  READ CARD-FILE RECORD AT END GO TO P3.
     READ RELATIVE-FILE RECORD INVALID KEY GO TO P4.
     IF DIGIT = 1 MOVE CARD-RECORD TO FILE-RECORD
     GO TO P5 ELSE IF DIGIT = 2 GO TO P6
     ELSE GO TO P7.
P3.  CLOSE CARD-FILE RELATIVE-FILE.
     STOP RUN.
P4.  DISPLAY 'KEY OUT OF RANGE' GO TO P2.
P5.  REWRITE FILE-RECORD INVALID KEY GO TO P4.
     GO TO P2.
P6.  ENTER LINKAGE.
     CALL 'SQUARES' USING I1 I2 I3.
     ENTER COBOL.
     GO TO P5.
P7.  ENTER LINKAGE.
     CALL 'CHANGE' USING I1 I2 TEMPORARY
     ENTER COBOL.
     GO TO P5.
```

Source Subprogram Statements (SQUARES)

```
ENTER LINKAGE.
ENTRY 'SQUARES' USING A B C.
COMPUTE C = A ** 2 + B ** 2.
RETURN.
ENTER COBOL.
```

Source Subprogram Statements (CHANGE)

```
ENTER LINKAGE.
ENTRY 'CHANGE' USING A B D.
MOVE A TO D.
MOVE B TO A.
MOVE D TO B.
RETURN.
ENTER COBOL.
```

In this Solution Program each record of RELATIVE-FILE, the 2311 file whose ORGANIZATION IS RELATIVE, is read, processed, and rewritten (so long as no INVALID KEY condition occurs). The REWRITE verb is used for this last step. REWRITE is used in the same manner as WRITE, but only for files opened as I-O and for records whose size is not altered between READ and REWRITE.

16.8 SPECIAL ROUTINES FOR ERROR-PROCESSING DECLARATIVE SECTIONS

The INVALID KEY option may be used for files to which ACCESS IS RANDOM when no error-processing declarative sections (see below) have been specified. The statements following INVALID KEY are executed when

one of the following conditions occurs:

1. When ORGANIZATION IS INDEXED and ACCESS IS RANDOM and when no record exists whose RECORD KEY field matches the contents of the SYMBOLIC KEY field.
2. When ORGANIZATION IS RELATIVE and ACCESS IS RANDOM and when the relative record number given in the SYMBOLIC KEY field is outside the limits of the file.
3. When ORGANIZATION IS DIRECT and ACCESS IS RANDOM and when the record is not found within the search limits of the file, or when the relative track number in the ACTUAL KEY field is outside the limits of the file.

Declarative sections provide an alternative way of specifying certain error-processing procedures. Such sections, if used, must be grouped together at the beginning of the PROCEDURE DIVISION and must follow the general form

```
PROCEDURE DIVISION.
DECLARATIVES.
(Section name) SECTION. USE (sentence).
(Procedural paragraphs).
END DECLARATIVES.
```

One or more declarative sections, whose names are programmer-selected, are written between DECLARATIVES and END DECLARATIVES, both of which must begin in Margin A of the coding form. The function of each declarative section is designated by the USE sentence, which may be one of five types, and the procedures to be executed are specified by a procedural paragraph at the programmer's option. The five types of USE sentence are:

1. USE FOR CREATING $\begin{matrix}\text{BEGINNING}\\\text{ENDING}\end{matrix}$ LABELS ON OUTPUT (*file name*).

   This sentence provides the capability for special label-processing procedures, BEGINNING for header labels and ENDING for trailer labels. If both these words are omitted, the declarative section processes both header and trailer labels.

2. USE FOR CHECKING $\begin{matrix}\text{BEGINNING}\\\text{ENDING}\end{matrix}$ LABELS ON INPUT (*file name*).

   This sentence provides for INPUT and I-O files the same label-processing abilities as the previous sentence provides for OUTPUT files.

3. USE AFTER STANDARD ERROR PROCEDURE ON (*file name*).

   This sentence allows the addition of more extensive input/output error-processing procedures than would ordinarily be written in connection with an INVALID KEY.

4.-5. USE FOR RANDOM PROCESSING and USE BEFORE REPORTING (*field name*) properly belong in the next chapter and are merely noted there.

16.9 OTHER COBOL FEATURES

There are other COBOL features which, though not illustrated, should be described within this chapter. First, there are the COBOL words; SAME, RERUN, REDEFINES, JUSTIFIED RIGHT, and EXIT.

16.9.1 SAME

SAME is an optional clause of the I-O-CONTROL paragraph. Its general form is

SAME AREA FOR (*File-name*-1 *File-name*-2 . . .).

This clause specifies that the files named are to use the same area when brought into memory for processing. Its function, therefore, is to conserve memory space, but its usefulness is limited to those programs in which it makes sense for only one of the files to be open at a time. More than one SAME clause may appear in a COBOL program, but a given file name may not appear in more than one such clause. Whenever a file named in a SAME clause is organized sequentially, the RESERVE clause must also be used.

16.9.2 The RERUN Clause

RERUN is also an optional clause of the I-O-CONTROL paragraph. Its form is

RERUN ON (*File-name*-1) EVERY END OF $\begin{matrix}\text{REEL}\\\text{UNIT}\end{matrix}$ OF (*File-name*-2).

This clause specifies, for large files which extend over several reels of tape or over several units of demountable disk storage, that *checkpoint* records are to be written every end of reel or unit. A checkpoint record is a recording of the status of the computer at a given point in the execution of the object program; it contains all of the information necessary to restart the program from that point. Checkpoint/restart procedures are used with long processing runs to avoid the necessity of restarting such runs from the beginning in the event some unanticipated interruption occurs; restarting and reprocessing are required only from the last checkpoint in such a case.

File-name-1 above is the name, as given in the name field of the DD control card, of the file on which checkpoint records are to be written; *File-name*-2 is the programmer-designated name of the file being processed. The latter must not be the name of a file opened as an I-O file, a file whose ORGANIZATION IS INDEXED, or a file for which ACCESS IS RANDOM. The above form of the RERUN clause will be generalized in the next chapter.

16.9.3 The REDEFINES Clause

REDEFINES is an optional clause that may be used in the description of a field. Its general form is

(*Field-name*-1) REDEFINES (*Field-name*-2)

The REDEFINES clause, like the SAME clause above, is designed to conserve memory. The fields specified in the REDEFINES clause are assigned the same memory addresses. When used, the REDEFINES clause must be the first clause following the field name that defines the entry.

16.9.4 The JUSTIFIED RIGHT Clause

JUSTIFIED RIGHT is also an optional clause that may be used in the description of a field. It may be written only for alphabetic and alphameric fields that are not subdivided and is written as

JUSTIFIED RIGHT

When a field of nonnumeric data is moved to a field for which JUSTIFIED RIGHT has been specified, the rightmost character of the source field is placed in the rightmost position of the receiving field. If the source field is longer than the receiving field, the extra characters are lost; if the source field is shorter than the receiving field, the extra positions are filled with blanks.

16.9.5 EXIT

EXIT is both a verb and an entire statement. When used, it must appear in the PROCEDURE DIVISION as a one-word paragraph preceded by a paragraph name. It is used, in an analogous way to the CONTINUE statement in a FORTRAN DO loop, when it is necessary to provide an end point for a procedure that is to be executed by means of a PERFORM statement or that is specified in a declarative section.

16.9.6 Variable Length Records

Finally, a number of comments about record formats must be made. It should be noted that all programming examples given in both this and the previous chapter have used fixed-length file records only. Thus the clause RECORDING MODE IS F has been used in every case, and the RECORD CONTAINS clause has in each case included a fixed number of characters only.

Variable-length records may be processed by specifying RECORDING MODE

IS V, by including the minimum and the maximum record sizes in the RECORD CONTAINS clause (for example, RECORD CONTAINS 150 TO 340 CHARACTERS), and by preceding each logical record in the file by a control word that indicates the size of the particular logical record. This control word cannot be described in any record description entry and cannot be referenced by the programmer.

Unspecified-length records may be processed by specifying RECORDING MODE IS U and by using the extended form of the RECORD CONTAINS clause given in the previous paragraph. Files with format U records are considered by the COBOL compiler to contain one logical record per block. The READ statement makes one block available for processing; if there are multiple logical records per block, the programmer must do his own deblocking.

When files are sequential or indexed and when their records are referenced only by the WRITE statement with FROM option, a special form of the APPLY clause may be written in the I-O-CONTROL paragraph. This is the clause whose form is

APPLY WRITE-ONLY ON (*File name*)

It is used to make optional use of buffer space allocated for a file whose record format is V, and is also used to make optimal use of device space when creating sequentially organized files whose record format is U. Normally, a buffer is truncated when the maximum size record no longer fits. Use of this clause will cause a buffer to be truncated only when the next record does not fit in the buffer.

PROBLEMS

1. What types of file organization are possible with COBOL? Why is it desirable to have different types?
2. What is the difference between sequential and random access to a file? When would each ordinarily be used?
3. What do the SAME clause, REDEFINES clause, and LINKAGE SECTION have in common? When is this a desirable property?
4. Which clause provides the ability to define data arrays? What is the maximum number of dimensions such arrays may have?
5. What subprogram capabilities are provided in COBOL? Compare these with the subprogram capabilities provided in FORTRAN.

A reel of magnetic tape contains 80-byte logical records in blocks of 20. These records are arranged in numerical sequence according to a 5-digit packed-decimal field in positions 78–80 of each record.

6. Write a program to copy this file from the magnetic tape to a 2311 unit, retaining the same blocking factor and both sequential organization and access.
7. Write a program to convert the above file from its sequential organization on magnetic tape to indexed organization on a 2311.

8. The records on the magnetic tape are in sequence according to the number in positions 78-80 of each; they are in random order, however, according to a 5-digit packed-decimal field in positions 75-77. Write a program to convert this file into a random-access file, based on the field in positions 75-77, and assuming that the value of that field never exceeds the available 2311 capacity.
9. Write a program to copy the file from magnetic tape to a 2311 unit, retaining sequential access, but converting the organization to direct.
10. Write a program to convert the file from sequential organization on magnetic tape to direct organization on a 2311 unit. Use the field in positions 75-77 as the track number, assumed never to exceed the number of available tracks, and use the field in positions 78-80 as the record number. Ensure that the field in positions 78-80, as used, is never greater than 19 by computing its value, modulo 19. Then add 1 to the result to produce a record number in the range 1-20.
11. Write a program both to count the number of records on the magnetic tape and to sum the contents of the field in positions 75-77. Then use a subprogram to compute and display the arithmetic mean of this field.
12. Write a program to compute and to print the square root of the packed-decimal number in positions 1-8 of each record on the magnetic tape. Do this by writing a Newton's method square-root procedure and referencing it by means of the PERFORM statement.
13. Use the TRANSFORM statement to alter positions 78-80 of each magnetic tape record such that, according to this field, the records will be in descending rather than ascending sequence.
14. A file of data cards has sixteen 5-digit integers punched in each. Assume the first eight are elements of one vector and the remaining eight are elements of another. Write a program to compute and to print the scalar product of vectors on each card. Use the OCCURS clause and subscripts.
15. A 2311 file with relative organization contains space for 8,000 records. Write a subprogram that accepts a 6-digit argument and compresses it into a number in the range 1-8,000.

Chapter 17

COBOL Advanced Features

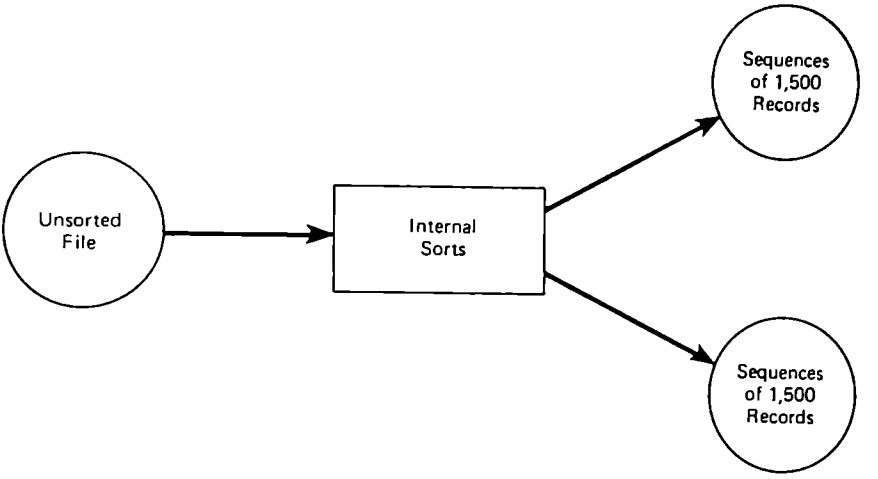
The advanced features discussed in this chapter are presented in four groups: (1) sorting, (2) report writing, (3) asynchronous processing, and (4) miscellaneous advanced features. The second of these, report writing, is of course a capability that must be present at every level of every usable programming system; included here are those special features that greatly facilitate the report-writing process for the COBOL programmer who has access to a large System/360 (>64 K).

17.1 SORTING

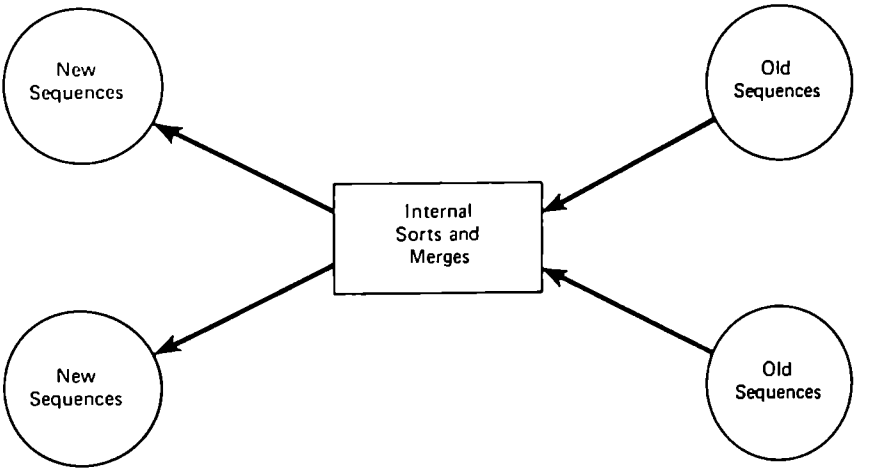
While there are too many specific sorting techniques to examine individually, the general procedure can be readily explained. Consider for example a file containing several hundred thousand records, only 1,500 of which can fit in memory simultaneously, and assume each record contains an index (or keyword or control word) which forms the basis for the desired sequence. Assume further, since it has previously been demonstrated, that it is possible to sort a group of records all of which are in memory. Then sorting of the entire file can be accomplished by the following procedure:

Step 1. Read 1,500 records into memory; sort internally and write out on an output device. Repeat, writing the results on an alternate output

device. Continue, alternating the use of the two output devices. Schematically,



Step 2. Using the output of the previous step as input, merge the two subfiles; that is, combine records from both subfiles wherever possible in the desired sequence, such that 1,500-record sequences are formed, and write them out alternately as before. Schematically,



Step 3. Repeat step 2 as many times as required to place all the records in the required sequence. Note that *previous step* refers to step 1 only the first time step 2 is executed; thereafter, it refers to the previous execution of step 2.

17.1.1
The SORT Feature in COBOL

Because sorting capabilities constitute a widespread requirement for the users of data-processing installations, sorting programs are ordinarily provided by the computer manufacturers; such programs are considered as essential complements of any system. As one would expect, therefore, sorting programs are provided for use with System/360, along with programming systems and other system support.

Sorting requirements do not ordinarily exist in and of themselves, however; the overall application of which they are a part usually contains a variety of other data processing operations. To the programmer responsible for creating an integrated solution to a problem it makes sense to include sorting capabilities within the programming system he is likely to be using when faced with sorting requirements. Such capabilities, therefore, have been included within COBOL.

The implementation of a sort feature within COBOL is accomplished primarily through the definition of a new type of file, the sort file, and through the use of the SORT statement in the PROCEDURE DIVISION.

17.1.2
The SORT Statement

The basic form for the SORT statement is

SORT *file* $\begin{matrix}\text{ASCENDING}\\\text{DESCENDING}\end{matrix}$ *v1*.

where *file* is a sort file named in the FILE CONTROL portion of the IDENTIFICATION DIVISION (by means of a SELECT statement) and described in the DATA DIVISION using the SD (sort description) entry. RECORD CONTAINS and/or DATA RECORD clauses are included as required.

v1 is the name of the key to be used for the sorting criterion. ASCENDING will cause the sorted records to be sequenced from lowest to highest key value, whereas DESCENDING will arrange them in the opposite order.

More than one sort key may be specified. Each must be preceded by ASCENDING or DESCENDING, with the most significant key being specified first. For example, if we are sorting hospital admission records (HOSPFILE) by hospital number (called HOSP) in increasing order and, for each hospital, by admission date (DATE-IN) starting with the most recent, and alphabetically by last name (LAST) for each date we would write

```
SORT HOSPFILE ASCENDING HOSP DESCENDING DATE-IN ASCENDING LAST.
```

The SORT statement may also include a phrase which names a section specifying any processing that is to be done prior to sorting (such as reading a record and determining whether or not it is to be included in the sorting

process). This is done by means of the INPUT PROCEDURE phrase which follows the last ASCENDING or DESCENDING phrase:

SORT *file* $\begin{matrix}\text{ASCENDING}\\\text{DESCENDING}\end{matrix}$ *v*1 $\begin{matrix}\text{ASCENDING}\\\text{DESCENDING}\end{matrix}$ *v*2 . . . . INPUT PROCEDURE *section*1

where *section*1 is the name of a section in the PROCEDURE DIVISION in which the preprocessing operations are stated. This section starts with

*section*1 SECTION.

and ends with

RELEASE *recname*.

where *recname* is the 01-level record name in the record description.

If no preprocessing is to be done, a USING clause must appear in place of the INPUT PROCEDURE clause. The form is

USING *infile*

When some type of processing is required for the sorted records, the SORT statement is augmented by an OUTPUT PROCEDURE clause which reads

OUTPUT PROCEDURE *section*2

No RELEASE is necessary for the corresponding processing section, since that statement interacts only with the sort program. Instead, it begins with a RETURN, which obtains properly sequenced records from the sort program. If no postprocessing is called for, the OUTPUT PROCEDURE is replaced by a phrase which says

GIVING *outfile*.

EXAMPLE 87

Each record in a large file consists of 20-character alphameric fields A and B, a 10-digit field C, a 15-character alphameric field D, 5-digit fields E and F, and 925 other characters. Write a program to do the following:

1. From those records in which fields A and B are not equal, construct new records consisting of fields A–F.
2. Sort the newly constructed records into sequence, using a composite key formed from fields A, B, and C. The proper sequence requires that field A end up in ascending order, field B in descending order, and field C be in ascending order.
3. After sorting, write those records in which fields E and F are equal on one file, and write field A of all other sorted records on another file.

Solution Program

Source Program Statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'SORTING'.
```

Source Program Statements

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360 F50.
OBJECT-COMPUTER. IBM-360 F50.
INPUT-OUTPUT SECTION.
FILE CONTROL.
  SELECT INPUT-FILE ASSIGN TO 'F401' UTILITY 2400 UNIT.
  SELECT SORT-FILE ASSIGN TO 'F402' UTILITY 2400 UNIT.
  SELECT OUT-FILE-1 ASSIGN TO 'F403' UTILITY 2400 UNIT.
  SELECT OUT-FILE-2 ASSIGN TO 'F404' UTILITY 2400 UNIT.
DATA DIVISION.
FILE SECTION.
FD INPUT-FILE BLOCK CONTAINS 5 RECORDS
  RECORD CONTAINS 1000 CHARACTERS
  RECORDING MODE IS F
  LABEL RECORDS ARE STANDARD
  DATA RECORD IS INPUT-RECORD.
01  INPUT-RECORD.
02  FIELD-A PICTURE X(20).
02  FIELD-B PICTURE X(20).
02  FIELD-C PICTURE 9(10).
02  FIELD-D PICTURE X(15).
02  FIELD-E PICTURE 9(5).
02  FIELD-F PICTURE 9(5).
02  FILLER PICTURE X(925).
SD SORT-FILE DATA RECORD IS SORT-RECORD
  RECORD CONTAINS 75 CHARACTERS.
01  SORT-RECORD.
02  A-FIELD PICTURE X(20).
02  B-FIELD PICTURE X(20).
02  C-FIELD PICTURE 9(10).
02  D-FIELD PICTURE X(15).
02  E-FIELD PICTURE 9(5).
02  F-FIELD PICTURE 9(5).
FD OUT-FILE-1 BLOCK CONTAINS 10 RECORDS
  RECORD CONTAINS 75 CHARACTERS
  RECORDING MODE IS F
  LABEL RECORDS ARE STANDARD
  DATA RECORD IS OUTPUT-1.
01  OUTPUT-1 PICTURE X(75).
  OUT-FILE-2 BLOCK CONTAINS 20 RECORDS
  RECORD CONTAINS 20 CHARACTERS
  RECORDING MODE IS F
  LABEL RECORDS ARE STANDARD
  DATA RECORD IS OUTPUT-2.
01  OUTPUT-2 PICTURE X(20).
```

Source Program Statements

```
PROCEDURE DIVISION.
  OPEN INPUT INPUT-FILE OUTPUT OUT-FILE-1 OUT-FILE-2.
  SORT SORT-FILE ASCENDING A-FIELD DESCENDING B-FIELD
  ASCENDING C-FIELD INPUT PROCEDURE RECORD-SELECTION
  OUTPUT PROCEDURE POST-PROCESS.
  CLOSE INPUT-FILE OUT-FILE-1 OUT-FILE-2.
  STOP RUN.
RECORD-SELECTION SECTION.
PARA-1. READ INPUT-FILE AT END GO TO PARA-2.
  IF FIELD-A = FIELD-B GO TO PARA-1 OTHERWISE
  MOVE FIELD-A TO A-FIELD MOVE FIELD-B TO B-FIELD
  MOVE FIELD-C TO C-FIELD MOVE FIELD-D TO D-FIELD
  MOVE FIELD-E TO E-FIELD MOVE FIELD-F TO F-FIELD
  RELEASE SORT-RECORD. GO TO PARA-1.
PARA-2. EXIT.
POST-PROCESS SECTION.
PARA-3. RETURN SORT-FILE AT END GO TO PARA-4.
  IF E-FIELD = F-FIELD WRITE OUTPUT-1 RECORD
  FROM SORT-RECORD GO TO PARA-3 ELSE
  WRITE OUTPUT-2 RECORD FROM A-FIELD
  GO TO PARA-3.
PARA-4. EXIT.
```

17.2 REPORT WRITING

Up until this point, report writing has not been considered different from writing an output file; conceptually, they are the same. Many installations, however, are heavily oriented toward the production of long, varied, and numerous printed reports, and have associated programming requirements that recur with high frequency. A special Report Writer feature has been included in COBOL, therefore, to simplify programming for such situations.

Without the Report Writer feature a COBOL programmer wanting his output in report format would have to specify that headings, footings, and individual fields be printed each time they are to appear; he has to keep track of the line count and page count, accumulate totals and provide for overflow situations where a page ends before an entire group of data has been printed.

With the Report Writer feature, headings and footings are described only once and are printed automatically at object time; line counters and page counters are incremented automatically; new pages are begun and page or overflow headings and footings are printed automatically; and data to be accumulated are summed and the totals are printed automatically.

The major entries in a COBOL program utilizing the Report Writer

feature are made in the DATA DIVISION. These consist of an entry in the FILE SECTION to list the names of the reports to be produced and of a special REPORT SECTION, which must be placed at the end of the DATA DIVISION, to define these reports. The REPORT SECTION describes the format of each report, with individual entries stipulating the following:

1. The maximum number of lines that can appear on a page.
2. The contents of the headings and footings, when they are required, and where they are to appear on the page.
3. The format of the data, the source of the data, and where they are to appear on the page.
4. The data to be summed, and when and where the totals are to appear on the page.

A report is produced by the execution of the INITIATE, GENERATE, and TERMINATE statements in the PROCEDURE DIVISION. INITIATE is used to initialize all counters associated with the REPORT WRITER. GENERATE is used each time a detail portion of the report is to be produced, and TERMINATE is used to end the report. In addition, a USE BEFORE REPORTING sentence is provided to allow additional processing via PROCEDURE DIVISION declaratives.

17.2.1 Definition of Reports

The FILE SECTION establishes the usage of a particular report name through a statement of the general form

FD *file* REPORT IS *reptname*

or

FD *file* REPORTS ARE *reptname1 reptname2* . . . .

if more than one report is to be produced from *file*.

17.2.2 The REPORT SECTION

A complete description of each report's physical layout is presented in the REPORT SECTION, placed at the end of the DATA DIVISION. For each report there are, in general, three types of entries: The Report Description entry (identified by RD), the Report Group Description entry (identified by level number 01), and the Report Element Description entry (identified by a level number in the range 02–49).

17.2.2.1 The Report Description Entry

This entry, which follows the statement

REPORT SECTION

contains RD in Margin A, followed by the name of the report to be produced.

(This corresponds to *reptname* in the file description). In addition, one or more of three optional clauses may be included:

1.

RD *reptname* CODE *'char'*

where *'char'* is a single character between quote marks. This is used in situations where several reports are placed on the same intermediate storage device (such as a single tape reel) prior to printing, with the records from these reports interspersed among each other. The CODE option causes COBOL to attach the specified character as a distinguishing tag so that the subsequent print program uses the proper records.

2.

RD *reptname* { CONTROL IS / CONTROLS ARE } [*v1* / FINAL / FINAL *v1*] *v1 v2* . . . .

where *v1*, *v2*, etc. are variables defined in the usual way. Any one of the three rightmost options may be used with either CONTROL IS or CONTROLS ARE.

The CONTROL clause is used when special actions are desired in conjunction with the change in value of specified variables. For example, the clause

CONTROLS ARE FINAL MONTH DAY

indicates that special actions are to be taken whenever the value of DAY changes, whenever the value of MONTH changes, and at the end of the report. DAY and MONTH are fields defined in the usual way, while FINAL is a COBOL word having a specific meaning in the REPORT SECTION. The special actions to be taken are defined by means of Report Group Description and Report Element Description entries.

3.

RD *reptname* PAGE *n* LINES HEADING *m*

The PAGE clause is used to describe the various printing boundaries for a report page. If the optional keyword LINES is included, it must be preceded by an integer *n* representing the maximum number of lines to be printed on the page. HEADING, also optional, must be followed by an integer *m*, indicating the number of the first line on which a heading may appear.

Additional options are

FIRST DETAIL *i*
LAST DETAIL *d*
FOOTING *f*

with the integers *i*, *d*, and *f* indicating, respectively, the first line on which a detail report group (part of the body of the report) may appear, the last line on which such an item may appear, and the 1st line on which a footing may appear.

Thus a complete Report Description could be

RD *reptname* CODE *'char'*. CONTROLS ARE FINAL v1 v2 PAGE *n* LINES HEADING *m* FIRST DETAIL *i* LAST DETAIL *d* FOOTING *f*.

17.2.2.2
The Report Group Description Entry

This entry describes a report group within a report. As such it follows the Report Description entry and must have an assigned level of 01. The 01 (always entered in Margin A) is followed by an optional variable name which identifies the report group. Although this name need not always be included, it must appear if that report group is to be referred to by a GENERATE statement in the PROCEDURE DIVISION.

Three clauses are available for use with this entry. Two of them (LINE and NEXT GROUP) are optional; the third (TYPE) is required.

1. The LINE clause, when used, specifies the line number on which the Report Group it is used with is to be printed. Such specification may be made in three ways. A clause such as LINE 40 means that the group is to be printed on the 40th line of the page; a clause such as LINE PLUS 2 means that the group is to be printed 2 lines after whatever was printed just prior to the group; and LINE NEXT PAGE means that the group is to be printed on the first detail line of the next page.
2. The NEXT GROUP clause is interpreted similarly to the LINE clause, but refers to line positioning *following* the printing of the associated group. For example, NEXT GROUP 23 means that the following group is to be printed on the 23rd line; NEXT GROUP PLUS 1 means that the next group is to be printed on the following line; and NEXT GROUP NEXT PAGE means that the next group is to be printed on the first detail line of the next page.
3. The TYPE clause is required in every Report Group Description entry; it specifies which of the nine kinds of Report Groups is being described. It always starts with the word TYPE which is followed either by the Report Group type or an abbreviation of such; in the case of CONTROL HEADING and CONTROL FOOTING, a further specification is made. The allowable types and abbreviations are as follows:

Report Group Type	*Abbreviation*
REPORT HEADING	RH
PAGE HEADING	PH
OVERFLOW HEADING	OH
CONTROL HEADING	CH

Report Group Type	*Abbreviation*
DETAIL	DE
CONTROL FOOTING	CF
OVERFLOW FOOTING	OV
PAGE FOOTING	PF
REPORT FOOTING	RF

REPORT HEADING (RH) is a Report Group that is printed only once, at the beginning of the report. A report may therefore have no more than one Report Group of this type.

PAGE HEADING (PH) is a Report Group that is printed at the beginning of each page. Again, there may be only one Report Group of this type in a report. When used, the PAGE clause must appear in the Report Description entry.

OVERFLOW HEADING (OH) is a Report Group that is printed at the beginning of the next page when the limitation set by LAST DETAIL (in the PAGE clause) has been exceeded.

CONTROL HEADING (CH) is followed by either a field name or the word FINAL. It is a Report Group that is printed, in the first case, when the value of the control field named changes and, in the second case, when the end of the report is reached.

DETAIL (DE) is a Report Group that is printed whenever reference to it is made by a GENERATE statement. This type of Report Group, which may appear several times within the same report, must be uniquely identified at the 01 level whenever used.

CONTROL FOOTING (CF) is similar to CONTROL HEADING, except for printing position. Both Report Groups are used to identify the sections and subsections within a report.

OVERFLOW FOOTING (OV) is similar to OVERFLOW HEADING, except for printing position. The former prints at the end of a page following recognition of an overflow condition, while the latter prints at the beginning of the following page.

PAGE FOOTING (PF) is similar to PAGE HEADING, except for printing position; REPORT FOOTING (RF) is similar to REPORT HEADING, except for printing position. Whereas a REPORT HEADING is printed only once in a report, at the beginning, a REPORT FOOTING is printed only once also, at the end.

17.2.2.3 The Report Element Description Entry

The Report Element Description entry must always begin with a level number in the range 02–49. This is followed by one or more of eleven subsequent items, according to the needs of the program. Of these items, several are familiar and/or have already been described. These are: variable

name, LINE clause, BLANK clause, PICTURE clause, VALUE clause, and JUSTIFIED RIGHT. Those to be described below are COLUMN clause, GROUP INDICATE, RESET clause, SOURCE clause, and SUM clause.

The COLUMN clause is used to indicate the position of the leftmost character of a field in a printline. For example, the entry

```
02  COLUMN 41 PICTURE A(9) VALUE 'CONTINUED'.
```

would cause CONTINUED to be printed in positions 41–49.

GROUP INDICATE (or just GROUP) is used when a field is to be printed only once following a control break or at the beginning of a new page. For example, it is often desired, when printing a column of monetary values, to print the dollar sign only with the first value in the column, at the beginning of a new page, and with the total.

The RESET clause is used when the programmer wants to override the automatic resetting of the sum counters after an associated control break: this clause is useful when cumulative totals are desired. It is written as RESET ON (or just RESET), followed by a field name or the word FINAL. RESET ON FINAL indicates that the counter is not to be reset until the final control footing has been printed.

Information is assembled by the Report Writer and prepared for output according to its description via the PICTURE clause. The SOURCE clause is used to locate the information to be processed. For example, the following entry specifies that the field denoted by MONTHNAME OF RECORD-AREA (MONTH) is to be printed starting in the 32nd position of the 45th line:

```
02  LINE 5 COLUMN 32 PICTURE A(9)
    SOURCE MONTHNAME OF RECORD-AREA (MONTH).
```

Finally, the SUM clause is used to specify the fields to be totaled and may only appear in a CONTROL FOOTING Report Group. The entry

```
02  COLUMN 30 PICTURE ZZ9 SUM NO-PURCHASES.
```

indicates with the printing information that the number of purchases is to be accumulated. A variation of the SUM clause is its usage with UPON. SUM PURCHASES UPON AREA-1, for example, carries out the totaling function only when the field named AREA-1 has been referenced by a GENERATE statement. This option may be employed, therefore, only when TYPE DETAIL has been included in the associated Report Group Description entry.

17.2.3 Report Writing Statements in the PROCEDURE DIVISION

The detailed report descriptions and specifications provide sufficient information to allow the actual report producing statements to be very simple. Three verbs are used for this purpose: INITIATE, GENERATE, and TERMINATE.

17.2.3.1 The INITIATE Statement

This statement, whose basic form is

INITIATE *reptname.*

initializes all counters associated with the production of *reptname.* The name *reptname*, as indicated before, is defined in the Report Description (17.2.2.1). When more than one report is being prepared by a single program, all their names may be given in a single INITIATE statement. When every report named in the REPORT SECTION is to be initialized, the simplified form

INITIATE ALL.

may be used. The file associated with a report must still be opened separately.

17.2.3.2 The GENERATE Statement

GENERATE, followed by the name of a variable defined in the REPORT SECTION, is the workhorse of the Report Writer feature. Not only is the field so named constructed and written out, but all special conditions associated with producing the specified report format are automatically handled. This includes the appropriate page or overflow footings and/or headings, the recognition of specified control breaks (that is, the ends of sections and subsections within the report), and the updating of counters as required.

17.2.3.3 The TERMINATE Statement

TERMINATE, like INITIATE, is followed either by one or more report names or by ALL. It ends the report by producing the terminal page and report footings.

EXAMPLE 88

Prepare a "Quarterly Expenditures Report" for the Acme Manufacturing Company (makers of manual and automatic acmes). Required input is on a tape file called INFILE, each logical record of which is called INPUT-RECORD. The file to be produced is called REPORT-FILE. Said file is to be prepared for offline printing. INFILE is blocked 25 records per block, each record containing 21 characters.

Solution Program

Source Program Statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'REPORT'.
```

Source Program Statements

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360 F50.
OBJECT-COMPUTER. IBM-360 F50.
INPUT-OUTPUT SECTION.
FILE-CONTROL
  SELECT INFILE ASSIGN TO 'F0023' UTILITY 2400 UNIT.
  SELECT REPORT-FILE ASSIGN TO 'F0101' UTILITY 2400 UNIT.
DATA DIVISION.
FILE SECTION.
FD INFILE BLOCK CONTAINS 25 RECORDS
  RECORD CONTAINS 21 CHARACTERS
  RECORDING MODE IS F
  LABEL RECORDS ARE STANDARD
  DATA RECORD IS INPUT-RECORD.
01  INPUT-RECORD
  02  FILLER PICTURE AA.
  02  DEPT PICTURE XXX.
  02  FILLER PICTURE AA.
  02  NO-PURCHASES PICTURE 99.
  02  FILLER PICTURE A.
  02  TYPE-PURCHASE PICTURE A.
  02  MONTH PICTURE 99.
  02  DAY PICTURE 99.
  02  FILLER PICTURE A.
  02  COST PICTURE 999V99.
FD REPORT-FILE REPORT IS EXPENSE-REPORT
  LABEL RECORDS ARE STANDARD.
WORKING-STORAGE SECTION.
01  FILLER.
  02  RECORD-MONTH
  03  FILLER PICTURE A(9) VALUE 'JANUARY  '.
  03  FILLER PICTURE A(9) VALUE 'FEBRUARY '.
  03  FILLER PICTURE A(9) VALUE 'MARCH    '.
  03  FILLER PICTURE A(9) VALUE 'APRIL    '.
  03  FILLER PICTURE A(9) VALUE 'MAY      '.
  03  FILLER PICTURE A(9) VALUE 'JUNE     '.
  03  FILLER PICTURE A(9) VALUE 'JULY     '.
  03  FILLER PICTURE A(9) VALUE 'AUGUST   '.
  03  FILLER PICTURE A(9) VALUE 'SEPTEMBER'.
  03  FILLER PICTURE A(9) VALUE 'OCTOBER  '.
  03  FILLER PICTURE A(9) VALUE 'NOVEMBER '.
  03  FILLER PICTURE A(9) VALUE 'DECEMBER '.
  02  RECORD-AREA REDEFINES RECORD-MONTH OCCURS 12 TIMES.
  03  MONTHNAME PICTURE A(9).
REPORT SECTION.
```

Source Program Statements

```
RD EXPENSE-REPORT CONTROLS ARE FINAL MONTH DAY
  PAGE 59 LINES HEADING 1 FIRST DETAIL 10 LAST DETAIL
  48 FOOTING 52.
01  TYPE REPORT HEADING.
  02  LINE 1 COLUMN 27 PICTURE A(26) VALUE IS
  'ACME MANUFACTURING COMPANY'.
  02  LINE 3 COLUMN 26 PICTURE A(29) VALUE IS
  'QUARTERLY EXPENDITURES REPORT'.
01  NEXT GROUP PLUS 1 TYPE PAGE HEADING.
  02  LINE 5 COLUMN 30 PICTURE A(9) SOURCE
  MONTHNAME OF RECORD-AREA (MONTH).
  02  COLUMN 39 PICTURE A(12) VALUE 'EXPENDITURES'.
  02  LINE 7 COLUMN 2 PICTURE X(35) VALUE IS
  'MONTH-----DAY----DEPT--NO-PURCHASES'.
  02  COLUMN 40 PICTURE X(33) VALUE IS
  'TYPE--------COST--CUMULATIVE-COST'.
01  NEXT GROUP PLUS 1 TYPE OVERFLOW HEADING.
  02  LINE 5 COLUMN 32 PICTURE A(9) SOURCE
  MONTHNAME OF RECORD-AREA (MONTH).
  02  COLUMN 41 PICTURE A(9) VALUE 'CONTINUED'.
  02  LINE 7 COLUMN 2 PICTURE X(35) VALUE IS
  'MONTH-----DAY----DEPT--NO-PURCHASES'.
  02  COLUMN 40 PICTURE X(33) VALUE IS
  'TYPE--------COST--CUMULATIVE-COST'.
01  DETAIL-LINE LINE PLUS 1 TYPE DETAIL.
  02  COLUMN 2 GROUP INDICATE PICTURE A(9) SOURCE
  MONTHNAME OF RECORD-AREA (MONTH).
  02  COLUMN 13 GROUP INDICATE PICTURE 99 SOURCE DAY.
  02  COLUMN 19 PICTURE XXX SOURCE DEPT.
  02  COLUMN 31 PICTURE Z9 SOURCE NO-PURCHASES.
  02  COLUMN 42 PICTURE A SOURCE TYPE-PURCHASES.
  02  COLUMN 50 PICTURE ZZ9.99 SOURCE COST.
01  LINE PLUS 2 NEXT GROUP PLUS 1 TYPE CONTROL FOOTING
DAY.
  02  COLUMN 2 PICTURE X(22) VALUE 'PURCHASES AND COST FOR'.
  02  COLUMN 24 PICTURE Z9 SOURCE MONTH.
  02  COLUMN 26 PICTURE X VALUE '-'.
  02  COLUMN 27 PICTURE 99 SOURCE DAY.
  02  COLUMN 30 PICTURE ZZ9 SUM NO-PURCHASES.
  02  MIN COLUMN 49 PICTURE $$$9.99 SUM COST.
  02  COLUMN 65 PICTURE $$$$9.99 SUM COST RESET ON FINAL.
01  LINE PLUS L NEXT GROUP PLUS 1 TYPE CONTROL FOOTING
MONTH.
  02  COLUMN 16 PICTURE A(14) VALUE 'TOTAL COST FOR'.
  02  COLUMN 31 PICTURE A(9) SOURCE
  MONTHNAME OF RECORD-AREA (MONTH).
```

Source Program Statements

```
  02 COLUMN 39 PICTURE AAA VALUE 'WAS'.
  02 INT COLUMN 46 PICTURE $$$9.99 SUM MIN.
01 NEXT GROUP PLUS 1 TYPE CONTROL FOOTING FINAL.
  02 LINE PLUS 1 COLUMN 16 PICTURE A(26) VALUE IS
  'TOTAL COST FOR QUARTER WAS'.
  02 COLUMN 45 PICTURE $$$$9.99 SUM INT.
01 TYPE OVERFLOW FOOTING.
  02 LINE 55 COLUMN 32 PICTURE A(9) SOURCE
  MONTHNAME OF RECORD-AREA (MONTH).
  02 COLUMN 41 PICTURE A(9) VALUE 'CONTINUED'.
  02 LINE 57 COLUMN 66 PICTURE X(5) VALUE 'PAGE-'.
  02 LINE 57 COLUMN 66 PICTURE X(5) VALUE IS 'PAGE-'.
01 TYPE PAGE FOOTING.
  02 LINE 57 COLUMN 66 PICTURE X(5) VALUE IS 'PAGE-'.
  02 COLUMN 70 PICTURE 99 SOURCE PAGE-COUNTER.
  01 TYPE REPORT FOOTING.
  02 LINE PLUS 1 COLUMN 32 PICTURE A(13) VALUE
  'END OF REPORT'.
PROCEDURE DIVISION.
  OPEN INPUT INFILE OUTPUT REPORT-FILE.
  INITIATE EXPENSE-REPORT.
READATA. READ INFILE AT END GO TO COMPLETE.
  GENERATE DETAIL-LINE.
  GO TO READATA.
COMPLETE. TERMINATE EXPENSE-REPORT.
  CLOSE INFILE REPORT-FILE.
  STOP RUN.
```

17.3 ASYNCHRONOUS PROCESSING*

Up to this point, not only with respect to COBOL but also with respect to both FORTRAN and Assembler Language, consideration has been given to synchronous processing only. By *synchronous processing* is meant the execution of instructions in a sequential manner. Synchronous processing has been, in fact, taken to be a concomitant of programming, with the execution of instructions being directly related to the line-by-line characteristic of the coding form.

Processing of large files stored on multiple direct-access devices is a situation in which the mechanical movement of access arms, the most time-consuming part of the operation, can very profitably be carried out in parallel. COBOL therefore, contains an advanced feature to facilitate the programmer's use of asynchronous processing for such purposes. By *asynchronous processing* is meant the execution of instructions in different seg-

*Currently not implemented.

ments of a program in such a way that no time-dependency exists between the segments and that allows a high degree of concurrent operation.

The facilities provided by COBOL for asynchronous processing include the saved area entry in the DATA DIVISION, the PROCESS, and HOLD verbs, the USE FOR RANDOM PROCESSING sentence in declarative sections, and a special form of the APPLY clause in the I-O-CONTROL paragraph of the INPUT-OUTPUT SECTION. These are illustrated in the example given below.

EXAMPLE 89

A transaction file on magnetic tape consists of 80-character records, five to a block. Each record contains a 10-digit account number in columns 11-20, a D (for deposit) or W (for withdrawal) in column 26, and an 8-digit amount (999999V99) in columns 32-39.

A master file on three 2311 units has DIRECT organization and consists of 77-byte unblocked records. Each record contains a 9-digit (S9999999V99) balance field, a 12-character name field, and a 60-character address field.

Write a program to update the balance fields in the master file by adding or subtracting the amount field of the transaction file, depending on whether the transaction is a deposit or withdrawal, respectively. Use the account number as the SYMBOLIC KEY; also use the account number to derive an ACTUAL KEY in the range 0-999.

Solution Procedure

Divide the PROCEDURE DIVISION into two parts, an *in-line* portion whose prime function is reading transaction records and transmitting them for subsequent asynchronous processing, and an *out-of-line* declarative section whose prime function is to perform this processing. The in-line procedure opens the files, reads a transaction record, moves it into the saved area (the special DATA DIVISION area provided for communication between in-line and out-of-line procedures), and initiates the out-of-line procedure by means of a PROCESS statement. It then reads the next transaction record. When the end-of-file condition is reached, the in-line procedure waits (by means of a HOLD statement) for the completion of any out-of-line procedures that are still being carried out, then closes the files and stops the run.

The out-of-line procedure (in the DECLARATIVES portion of the PROCEDURE DIVISION) calculates the ACTUAL KEY from the account number in the saved area, reads the appropriate record from the master file, updates the record, and rewrites it onto the file. An out-of-line HOLD is also used to force completion of the out-of-line cycles in the order in which they were initiated.

Solution Program

Source Program Statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'UPDATE'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360 F50.
```

Source Program Statements

```
OBJECT-COMPUTER. IBM-360 F50.
INPUT-OUTPUT SECTION.
FILE CONTROL
  SELECT TRANSACT ASSIGN TO 'F050' UTILITY 2400 UNIT.
  SELECT ACCOUNTS ASSIGN TO 'F051' DIRECT-ACCESS 2311 UNITS
  ACCESS IS RANDOM ORGANIZATION IS DIRECT
  SYMBOLIC KEY IS SA-ACCOUNT-NUMBER
  ACTUAL KEY IS SA-TRACK-NUMBER.
I-O-CONTROL.
  APPLY RANDOM-UPDATE TO TRANSFER-AREA ACCOUNTS FOR 8 CYCLES
  APPLY RECORD PROTECTION ON ACCOUNTS.
DATA DIVISION.
FILE SECTION.
FD TRANSACT RECORDING MODE IS F
  BLOCK CONTAINS 5 RECORDS
  RECORD CONTAINS 80 CHARACTERS
  LABEL RECORDS ARE STANDARD
  DATA RECORD IS TRANSACTION-RECORD.
01  TRANSACTION-RECORD.
  02  FILLER PICTURE X(10).
  02  ACCOUNT-NUMBER PICTURE X(10).
  02  FILLER PICTURE X(5).
  02  TRANSACTION-CODE PICTURE X.
  02  FILLER PICTURE X(5).
  02  AMOUNT PICTURE 9(6)V99.
  02  FILLER PICTURE X(41).
FD ACCOUNTS RECORDING MODE IS F
  RECORD CONTAINS 77 CHARACTERS
  LABEL RECORDS ARE STANDARD
  DATA RECORD IS ACCOUNT-RECORD.
01  ACCOUNT-RECORD.
  02  ACC-BALANCE PICTURE S9(7)V99 COMPUTATIONAL-3.
  02  ACC-NAME.
  03  LAST PICTURE X(10).
  03  FIRST-INIT PICTURE X.
  03  MIDDLE-INIT PICTURE X.
  02  ACC-ADDRESS.
  03  STREET PICTURE X(20).
  03  CITY PICTURE X(20).
  03  STATE PICTURE X(20).
SA  TRANSFER-AREA.
01  SA-REC.
  02  SA-TRANS-IN.
  03  ACCOUNT-NUMBER PICTURE X(10).
  03  SA-ACCOUNT-NUMBER REDEFINES ACCOUNT-NUMBER OF SA-REC.
  04  SA-AN1 PICTURE 9(4).
```

Source Program Statements

```
    04  SA-AN2 PICTURE 9(4).
    04  SA-AN3 PICTURE 9(2).
    03  TRANSACTION-CODE PICTURE X.
    88  DEPOSIT VALUE 'D'.
    88  WITHDRAWAL VALUE 'W'.
    03  AMOUNT PICTURE 9(6)V99.
    02  SA-W-S.
    03  SA-TRACK-NUMBER PICTURE S999 COMPUTATIONAL.
    03  SA-ERR-NO PICTURE 9.
    03  SA-WORK PICTURE S9999 COMPUTATIONAL.
    03  SA-OLD-BALANCE PICTURE $*,***,**9.99-.
    03  SA-NEW-BALANCE PICTURE $*,***,**9.99-.
PROCEDURE DIVISION.
DECLARATIVES.
RANDOM-UPDATE SECTION.
    USE FOR RANDOM PROCESSING.
OUT-OF-LINE.
    NOTE THIS IS THE START OF THE OUT-OF-LINE PORTION.
INITIALIZE.
    MOVE ZERO TO SA-ERR-NO.
CALCULATE-ACTUAL-KEY.
    IF ACCOUNT-NUMBER IN SA-TRANS-IN IS NOT NUMERIC
    MOVE 4 TO SA-ERR-NO GO TO FORCE-SEQUENTIAL.
    COMPUTE SA-WORK = SA-AN1 - SA-AN2.
    IF SA-WORK IS NEGATIVE
    COMPUTE SA-WORK = 9999 + SA-WORK ELSE
    COMPUTE SA-WORK = 9999 - SA-WORK.
    IF SA-AN3 = 0
    COMPUTE SA-TRACK-NUMBER = SA-WORK / 10 ELSE
    COMPUTE SA-TRACK-NUMBER = SA-WORK / (SA-AN3 * 10).
READ-ACCOUNT-RECORD.
    READ ACCOUNTS INVALID KEY MOVE 1 TO SA-ERR-NO
    GO TO FORCE-SEQUENTIAL.
    MOVE ACC-BALANCE TO SA-OLD-BALANCE.
UPDATE-BALANCE.
    IF DEPOSIT NEXT SENTENCE ELSE GO TO UPDATE-1.
    ADD AMOUNT IN SA-REC TO ACC-BALANCE ON SIZE ERROR
    MOVE 2 TO SA-ERR-NO GO TO FORCE-SEQUENTIAL.
    GO TO SAVE-RECORD.
UPDATE-1.
    IF WITHDRAWAL SUBTRACT AMOUNT IN SA-REC
    FROM ACC-BALANCE ELSE MOVE 5 TO
    SA-ERR-NO GO TO FORCE-SEQUENTIAL.
    IF AMOUNT IN SA-REC IS NEGATIVE MOVE 6
    TO SA-ERR-NO GO TO FORCE-SEQUENTIAL.
SAVE-RECORD.
```

Source Program Statements

```
    MOVE ACC-BALANCE TO SA-NEW-BALANCE.
    REWRITE ACCOUNT-RECORD INVALID KEY
    MOVE 3 TO SA-ERR-NO.
FORCE-SEQUENTIAL.
    HOLD RANDOM-UPDATE.
ERROR-CHECK.
    GO TO E1 E2 E3 E4 E5 E6 DEPENDING ON SA-ERR-NO.
    GO TO END-RANDOM.
E1. DISPLAY 'ACCOUNT IS NOT IN ACCOUNTS FILE.'.
    GO TO END-2.
E2. DISPLAY 'BALANCE EXCEEDS $9,999,999.99.'.
    GO TO END-1.
E3. DISPLAY 'INVALID KEY ERROR TRYING TO REWRITE ACCOUNTS.'.
    DISPLAY 'NEW BALANCE IS' SA-NEW-BALANCE.
    GO TO END-1.
E4. DISPLAY 'ACCOUNT NUMBER IS NOT NUMERIC.'.
    GO TO END-2.
E5. DISPLAY 'ILLEGAL TRANSACTION CODE IS SPECIFIED.'.
    GO TO END-2.
E6. DISPLAY 'BALANCE OVER-DRAWN.'.
    GO TO END-2.
END-1.
    DISPLAY 'OLD BALANCE IS' SA-OLD-BALANCE.
END-2.
    DISPLAY 'TRANSACTION IS' ACCOUNT-NUMBER
    IN SA-REC ' ' TRANSACTION-CODE IN SA-REC
    ' ' AMOUNT IN SA-REC.
END-RANDOM.
    EXIT.
END DECLARATIVES.
IN-LINE.
    NOTE THIS IS THE START OF THE IN-LINE PORTION.
START.
    OPEN INPUT TRANSACT I-O ACCOUNTS.
INITIATE-RANDOM-PROCESSING.
    READ TRANSACT AT END GO TO IN-LINE-HOLD.
    MOVE CORRESPONDING TRANSACTION-RECORD TO SA-TRANS-IN.
    PROCESS RANDOM-UPDATE.
    GO TO INITIATE-RANDOM-PROCESSING.
IN-LINE-HOLD.
    HOLD RANDOM-UPDATE.
    CLOSE TRANSACT ACCOUNTS.
    STOP RUN.
```

Despite the apparent complexity of the Solution Program, just a few statements give it its asynchronous processing capability. These are the two APPLY clauses in the I-O-CONTROL paragraph, the saved area (TRANSFER-AREA) in the DATA DIVISION, the declarative section (RANDOM-UPDATE) with its USE FOR RANDOM

PROCESSING sentence, the two HOLD RANDOM-UPDATE statements, and the PROCESS RANDOM-UPDATE statement.

As identified by NOTE statements, the PROCEDURE DIVISION is divided into an in-line segment and an out-of-line segment. Asynchronous processing is initiated by the execution of the PROCESS statement (the verb PROCESS followed by a section name) in the in-line segment. In the Solution Program, PROCESS RANDOM-UPDATE performs this initiation function.

Asynchronous processing initiated in the section so named is the section in a declarative section that contains the USE FOR RANDOM PROCESSING sentence. A declarative section used for asynchronous processing cannot contain STOP, OPEN, CLOSE, CALL, or PROCESS statements, cannot refer to paragraph names and section names outside of the declarative section, and must have all logical paths within the section leading to a common exit point from the section. Note the use of EXIT for this last purpose in the Solution Program.

The purpose of the HOLD statement, as used in the in-line portion of the program, is to insure that all asynchronous processing, previously initiated, has been completed before any subsequent statements are executed. In the Solution Program, for example, the files should not be closed and the run should not be stopped until processing has been completed on all previously read transaction records. HOLD may be followed by one or more section names or by the word ALL. The option chosen by the programmer determines which out-of-line sections, where there are more than one, are to be concluded before in-line processing is allowed to continue.

When used in an out-of-line portion of a program, HOLD must be followed by the name of the declarative section in which it appears. It causes out-of-line processing cycles to be processed in the order in which they were initiated. This capability can be useful, as in the Solution Program, to force asynchronous processing, so that any error messages produced will appear in the same order as the transactions were entered and to prevent intermixing lines of messages. Only one HOLD statement may be included in an out-of-line section, and the logic must be such that every path flows through such statement.

The special saved area (SA) entry in the DATA DIVISION is used to define a communications region, so that data may be passed from the in-line portion to the out-of-line portion. The SA entry is analogous to the FD and SD entries, but it is more limited. SA is followed by the name assigned to the area, then by an optional RECORD CONTAINS clause, and then by appropriate record description entries. Every saved area named in a program must appear in an APPLY . . . TO . . . FOR . . . CYCLES clause (see following paragraph).

In the Solution Program clause, APPLY RANDOM-UPDATE TO TRANSFER-AREA ACCOUNTS FOR 8 CYCLES, the out-of-line declarative section (RANDOM-UPDATE) is related to the saved area (TRANSFER-AREA) processed by the out-of-line procedure and to the file (ACCOUNTS) which is processed in an asynchronous fashion. FOR 8 CYCLES, an optional part of the clause, specifies the maximum number of asynchronous processing cycles that are to be active at one time in the out-of-line procedure. If this part of the APPLY clause is omitted, COBOL operates as if FOR 5 CYCLES had been specified.

APPLY RECORD PROTECTION ON, followed by one or more file names, specifies that an individual record of the file or files named, when processed in one cycle, is protected from manipulation by another cycle in an asynchronously processed program. When using this feature, a cycle that reads a record (by a READ statement)

must execute a REWRITE statement for the record before another READ statement can be executed. Also, in order to free a record from record protection, it is necessary to execute a REWRITE statement for each record that is read before the execution of any out-of-line HOLD statement. This step must be taken for each cycle, even when no alteration to the contents of the record is made.

Where the EXAMINE statement is used in an asynchronously processed program, a unique TALLY exists for each cycle of each out-of-line declarative section as well as for the in-line portion.

Illustrated in the Solution Program, though entirely unrelated to the asynchronous processing capability, is the MOVE CORRESPONDING statement. This form of the MOVE statement is equivalent to one or more ordinary MOVE statements. When this form is used, selected subfields within the first field named are moved to selected subfields within the second field name, the selection being based on matching the names of the subfields. Thus if we examine the composition of TRANSACTION-RECORD and SA-TRANS-IN, we can see that MOVE CORRESPONDING TRANSACTION-RECORD TO SA-TRANS-IN is equivalent to the three MOVE statements: MOVE ACCOUNT-NUMBER OF TRANSACTION-RECORD TO ACCOUNT-NUMBER OF SA-TRANS-IN, MOVE TRANSACTION-CODE OF TRANSACTION-RECORD TO TRANSACTION-CODE OF SA-TRANS-IN, and MOVE AMOUNT OF TRANSACTION-RECORD TO AMOUNT OF SA-TRANS-IN. ADD CORRESPONDING and SUBTRACT CORRESPONDING operate in the same way.

17.4 OTHER ADVANCED FEATURES

17.4.1 The SPECIAL-NAMES Paragraph

The CONFIGURATION SECTION of the ENVIRONMENT DIVISION may contain a paragraph headed SPECIAL-NAMES. This allows the COBOL programmer to specify names of his choice for various items in his program or the surrounding system.

1. The CODE clause in the Report Description Entry (Sec. 17.2.2.1) may be used in an alternate form to the one described previously. Instead of specifying the literal code in the Report Description, we can write

   RD *reptname* WITH CODE *codename*

   where *codename* is a variable name assigned to a single character in the SPECIAL-NAMES paragraph:

   SPECIAL NAMES.
   'char' IS *codename*.

2. The SPECIAL-NAMES paragraph may be used to associate SYSIN, SYSOUT, SYSPUNCH, or CONSOLE with a programmer-specified mnemonic name. For example, the sentence CONSOLE IS LOG-SHEET would allow the use of the name LOG-SHEET in the UPON clauses of the DISPLAY and ACCEPT statements of the PROCEDURE DIVISION. With the above sen-

tence specifying the name by which the console device is known to the compiler, we can write a statement such as DISPLAY CHECK-SUM UPON LOG-SHEET.

3. The SPECIAL-NAMES paragraph also provides a site for the DECIMAL-POINT IS COMMA clause. When this clause is written, the functions of the comma and period are interchanged in the PICTURE clause and in numeric literals, the purpose being to facilitate the use of COBOL where such a convention is used. Note, however, that this clause serves only to interchange the way COBOL treats commas and periods; it is the responsibility of the programmer to represent the decimal point, when required in a numeric literal or in the PICTURE clause, by a comma, and vice versa.

17.4.2 INPUT-OUTPUT SECTION Features

Three advanced features are related to the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION. The TRACK-AREA clause of the FILE-CONTROL paragraph, already encountered (Example 84), may specify an integer as the number of characters, in other words, (TRACK-AREA IS 2400 CHARACTERS).

A FILE-LIMIT clause may be included in the FILE-CONTROL paragraph to specify the number of tracks to be initialized for the creation of files with DIRECT organization (FILE-LIMIT IS 500 TRACKS).

The RERUN clause of the I-O-CONTROL paragraph, in addition to specifying that checkpoint records are to be written EVERY END OF REEL or EVERY END OF UNIT, may specify an integral number of minutes between the writing of successive checkpoint records such as (RERUN ON 'F040' EVERY 30 CLOCK-UNITS).

17.4.3 Additional CLOSE Statement Features

Several additional alternatives are provided in conjunction with the CLOSE statement. If neither the NO REWIND or LOCK is specified with UNIT or REEL, repositioning to the beginning of the file takes place unless the REVERSED option of the OPEN has been specified, in which case repositioning to the end of the volume occurs. If the NO REWIND option is specified in conjunction with the UNIT or REEL option, no repositioning takes place at the time the volume is closed. If the LOCK option is specified in conjunction with the UNIT or REEL option, the action taken (rewind, no rewind, lock) is a function of the DISP subparameter of the DD control card. If more than one CLOSE UNIT or CLOSE REEL for a given file is specified, all must have the same NO REWIND or LOCK option if one of these is specified.

In addition to the above CLOSE statement options, if neither UNIT nor REEL is specified, and the NO REWIND option is specified, positioning to the end

of the file being processed takes place. If the file is being processed with the REVERSED option, positioning takes place at the beginning of the file. The NO REWIND option should be written only for files assigned to magnetic tape and direct-access devices. If neither UNIT nor REEL is specified, and the LOCK option is specified, the action taken is a function of the DISP subparameter of the DD Control card.

17.4.4 Expanded IF Statements

An advanced feature that allows logical statements in COBOL to make a closer approximation to the corresponding English statements is also present. This is the use of implied subjects and operators in certain cases. The statement, IF A IS GREATER THAN B AND A NOT LESS THAN C GO TO Z, for example, may be shortened to: IF A IS GREATER THAN B AND NOT LESS THAN C GO TO Z. Also, the statement, IF A IS GREATER THAN B AND A IS GREATER THAN C GO TO Z, may be shortened to: IF A IS GREATER THAN B AND C GO TO Z. This capability may be used wherever repetition of a subject or an operator in a parallel structure occurs.

17.4.5 COBOL Debugging Language

A number of statements are provided in the COBOL language to help the programmer debug his program. They may appear throughout a source program or may be grouped together into a *debugging packet* which is a group of statements preceded by a special control card and placed either just prior to or just after the source statements, depending on the COBOL compiler being used. The special control card contains *DEBUG in columns 1-6, the name of the PROCEDURE DIVISION section or paragraph where the packet is to be executed starting in column 8, and, optionally, the word TRY following this name and separated from it by a comma. When a program is being both compiled and executed, the TRY option indicates to the compiler that errors found within the debugging packet should not prevent the main program from being executed.

A debugging packet may consist of any COBOL procedural statements. In addition, the special statements provided are as follows:

1. READY TRACE
   After this statement is encountered, each time execution of a paragraph or section begins, a message to that effect is written on the device defined as SYSOUT.
2. RESET TRACE
   This statement is used to terminate the execution of READY TRACE. Pairs of these statements may be used to produce highly selective insight into the way specific procedures are carried out.

3. EXHIBIT NAMED (followed by one or more field names and/or nonnumeric literals). Whenever this statement is executed, a message is written on SYSOUT. This message consists of, for each field named or literal specified, the field name, an equals sign, and the value of the field, all suitably separated by blanks. The sum of the sizes of the operands of an EXHIBIT statement cannot exceed the SYSOUT maximum logical record length.
4. EXHIBIT CHANGED (followed by one or more field names)
   This statement is similar to EXHIBIT NAMED, with the following exception: a field whose value has not changed since the last execution is left blank.
5. EXHIBIT CHANGED NAMED (followed by one or more field names) This statement is similar to EXHIBIT NAMED, with the following exception: a field whose value has not changed since the last execution is omitted.
6. ON (followed by a count-condition and then by the same structure that follows a condition in an IF statement)
   A *count-condition* is a clause that specifies the starting value, increment size, and final value of a counter that can only count integers. The count-condition, 10 AND EVERY 2 UNTIL 50, for example, sets a counter at 10 and increments it by 2 until the value of 50 is reached. The count-condition is incorporated into the ON statement as follows:

```
ON 10 AND EVERY 2 UNTIL 50 GO TO SPECIAL-PROCEDURE ELSE
NEXT SENTENCE.
```

   The 10th, 12th, 14th, etc. time this statement is encountered, the branch to SPECIAL-PROCEDURE is taken; all other times control goes to NEXT SENTENCE. Whenever AND EVERY (followed by the second integer) is omitted, the increment is taken to be 1. Whenever AND through the third integer is omitted, the statement prior to ELSE is executed but once.

17.4.6 Source Program Library Facility

At most installations where COBOL is used, it is advantageous to a programmer to be able to include prewritten source-program segments in his program at the time of compilation. Thus, COBOL provides him with the capability of using standard file descriptions, record descriptions, or procedures without having to repeat programming them. These program segments are contained in an installation-created library and are referenced by installation-assigned names, such names following the same specifications as COBOL program names.

COBOL's Source Program Library Facility includes a COPY clause that allows programmer access to prewritten ENVIRONMENT DIVISION and DATA DIVISION entries, an INCLUDE statement that allows access to PROCEDURE

DIVISION entries, and a BASIS card that allows access to and modification of complete programs.

The COPY clause consists of COPY followed by the installation-library name of the coding to be copied. The placement of this clause, which depends upon the coding being copied, is as follows:

1. Within the CONFIGURATION SECTION, the COPY clause is written in the SPECIAL-NAMES paragraph.
2. Within the INPUT-OUTPUT SECTION, the COPY clause may appear after FILE-CONTROL or I-O-CONTROL to utilize a whole paragraph or may appear after the file name in the SELECT sentence.
3. Within the FILE SECTION, the COPY clause may appear after the file, sort file, saved area, and/or report name to utilize a complete description or may appear after the fields named in 01 or 77-level entries to utilize a specific segment.

The INCLUDE statement is similar in format to the COPY clause and is used in a similar way, following paragraph or section names according to whichever type of entity is being utilized. For example, the entry, PARA-1. INCLUDE 'LP017'., assigns the paragraph name, PARA-1, to the paragraph named LP017 in the Source Program Library and compiles that paragraph into the program of which this entry is a part.

A complete program may be called from the Source Program Library by means of the BASIS card. This card contains BASIS in columns 1–5 and the name of the desired program starting in column 8. If it were desired to use an entire program without alteration, the object-language version would ordinarily be selected. Selection of an entire source language program would rarely be done unless it were desired to make source language alterations to it. COBOL provides, along with the BASIS card, therefore, the capability of deleting statements from and adding statements to the program selected by such card.

To delete a statement of the source program, a DELETE card is placed behind the BASIS card. It is punched DELETE in columns 1–6 and has the statement sequence number from the compilation listing starting in column 8. Deletions of several consecutive statements may be made with a single DELETE card by punching the beginning and ending sequence numbers, separated by a hyphen. Deletions of nonconsecutive statements or of several ranges of statements may be made with a single DELETE card by separating the entries with a comma and a space; the columns 8–72 are available for such usage.

To insert a statement into the source program, an INSERT card is placed behind the BASIS card. This card is punched INSERT in columns 1–6 and has a single sequence number starting in column 8. Immediately behind the INSERT card must be one or more COBOL statements that are to be inserted after the statement whose sequence number was punched in the INSERT card.

17.4.7
Sterling Currency Feature

Although it will not be described in this text, the capability of COBOL for handling sterling currency should be noted. COBOL provides the ability through the PICTURE clause to accept sterling data punched according to either the British Standards Institution method or the IBM method, to process such data (whose definition also includes the expanded form, USAGE IS DISPLAY-ST), and to write properly formatted reports incorporating the same degree of flexibility as has been described for dollars-and-cents items.

PROBLEMS

1. Of the advanced features included in this chapter, which has the greatest utility for programmers, generally. Which has the greatest utility for the special needs of COBOL programmers?
2. Considering the types of applications for which COBOL is ordinarily employed, what is likely to be the most-used Source Program Library Facility feature by COBOL programmers?
3. What is the advantage of COBOL's asynchronous processing capability? Would such a capability be an advantage in FORTRAN?
4. In what way might COBOL's sorting and report writing capabilities be combined in the same program?
5. Which language is more powerful, COBOL or FORTRAN? Why?

A large file stored on magnetic tape consists of 80-byte records in blocks of 20. Each record has the following description:

```
01  STUDENT-RECORD.
    02  STUDENT-NAME.
        03  LAST-NAME PICTURE X(18)
        03  INITIALS PICTURE X(2).
    02  FILLER PICTURE X(56).
    02  SEMESTER-HOURS PICTURE 999 COMPUTATIONAL-3.
    02  CUMULATIVE-AVERAGE PICTURE 9V99 COMPUTA-
        TIONAL-3.
```

6. Write a program to sort this file by STUDENT-NAME.
7. Indicate what changes should be made in the above program to produce a listing of student names and cumulative averages as a by-product of sorting, for each student whose cumulative average is 2.50 or greater.
8. Indicate what changes should be made in the program of Exercise 6 to sort the file by STUDENT-NAME within CUMULATIVE-AVERAGE.
   (The cumulative averages should be in descending order, and the student records having the same average should be in ascending order by STUDENT-NAME).

9. On a large sheet of paper ruled so as to have at least 59 lines and 72 columns, insert entries of a typical page of the report of Example 7.
10. Using another sheet of paper ruled in the same way, plan a STUDENT-STANDING-REPORT which shows for each CUMULATIVE-AVERAGE the names of students having that average and also the total number of students having that average. Write a program to produce this report, using the sorted file of Exercise 8 as input, and printing the report on a 1403.
11. A deck of cards contains each student's last name in columns 1–18, his initials in columns 18–20, the number of semester hours completed during the past semester in columns 76–77, and his academic average of the past semester in columns 78–80. Write a program to update the sorted file in Exercise 6. What has to be done before this program can be executed?
12. Assume that the file of student records above is stored on direct access 2311 units, with DIRECT organization and with a 10-digit student number in positions 21–30. Assume also that the deck of cards above does not have the student's names and initials punched but the student numbers in columns 1–10 instead. Write a program in COBOL to update the student record file. Use the student number as the SYMBOLIC KEY and derive the ACTUAL KEY from the student number, assuming the tracks in the file are numbered 000–599.
13. Indicate what change should be made in the program of Exercise 12 to help the programmer test that part of the program in which the ACTUAL KEY is computed.
14. Indicate what change should be made in the program of Exercise 12 to enable the program to check the cumulative average updating calculation for each hundredth student.
15. Indicate what change should be made in the program of Exercise 12 to enable the programmer to take advantage of a preprogrammed cumulative average procedure in the Source Program Library. This procedure, which is named CAP, is written in such a way that the following calculations are carried out: CUMULATIVE-AVERAGE * CUMULATIVE-HOURS + NEW-HOURS and the result replaces CUMULATIVE-AVERAGE; and CUMULATIVE-HOURS + NEW-HOURS replaces CUMULATIVE-HOURS.

Chapter 18

Programming in the PL/I Language

Both the FORTRAN and COBOL languages were created for use with computers which predated System/360 by several years. Compilers for these languages have been provided for use with System/360 for two reasons. First, continuity of language makes conversion to System/360 easier. Second, many computer users are satisfied with FORTRAN and COBOL.

PL/I (for Programming Language/One), created especially for System/360, was introduced for a number of well-founded reasons:

1. FORTRAN is oriented toward algorithmic procedures and COBOL toward the maintenance of files and the production of reports; neither is a general-purpose language. System/360 is a general-purpose computer, however, and is used not only in ways requiring the capabilities of both FORTRAN and COBOL but also in ways for which neither FORTRAN nor COBOL is suited. Thus the first reason for the creation of PL/I is to provide a single language with the ability to handle all sorts of programming requirements.
2. Generally, both FORTRAN and COBOL, even for simple applications, require the programmer to master a certain body of knowledge, and such mastery enables the programmer to use a certain subset of System/360 capabilities. This is an imposition on the programmer which PL/I attempts to correct. The language is structured in such a way

that the user whose problems require little sophistication need learn only a part of the language, whereas the programmer whose problems require the full power of System/360 has available to him almost as much as through Assembler Language.

3. Although both FORTRAN and COBOL enable the programmer to specify problem solutions in a language akin to that in which the problems are normally stated, both languages have numerous syntactic requirements wholly unrelated to such problems. Most of these requirements facilitate compiler design and improve compilation speed. Taking advantage of the inherent speed of System/360, PL/I provides the problem programmer with a simplified syntax.
4. Aside from the inherent technical and logistic advantages of a "universal" high-level language, there are some conspicuous economic ones as well. It is becoming increasingly expensive to support and maintain two separate languages. This is true for both the users and suppliers. Consequently, a growing faction in the computer community is advocating the eventual replacement of FORTRAN and COBOL.

As PL/I is described, it will be noted that it uses many of the words used in FORTRAN and COBOL. Some of these (such as GO TO) are used in the same way, while others (such as END) are used in an almost entirely different way. Therefore, no assumptions as to carryover should be made.

18.1 BASIC SYNTAX AND CODING RULES

PL/I is not designed with any particular input device in mind (although the punched card is presently the most common medium). Consequently there is no correspondence between the length or format of a PL/I statement and the available number of card columns. Instead, there is great emphasis on flexibility of formats and relative freedom from syntactic constraints. One property that has received much attention is the *default mechanism*. To a much greater extent than was true elsewhere, PL/I's basic design stresses its ability to act on incomplete specifications by making "reasonable" assumptions with regard to implied items of information. We shall run into this repeatedly.

18.1.1 PL/I Character Sets

Although PL/I programs will accept any and all 256 types of characters as data, only certain subsets of these may be used to write PL/I statements. Two separate character sets are accepted: A restricted 48-character set, oriented to the older model (26) keypunch and a 60-character set (which we shall use in this text) consistent with the later model (29). These are compared in Table 18.1. Certain items represented by single symbols in the 60-character set are replaced by two-character combinations in the

TABLE 18.1 PL/I CHARACTER SETS

Symbol in 60-character Set	Equivalent in 48-character Set	Symbol in 60-character Set	Equivalent in 48-character Set	Symbol in 60-character Set	Equivalent in 48-character Set
A	A	U	U	*	*
B	B	V	V	/	/
C	C	W	W	(	(
D	D	X	X	)	)
E	E	Y	Y	'	'
F	F	Z	Z		
G	G	0	0	.	.
H	H	1	1	%	//
I	I	2	2	,	,
J	J	3	3	:	:
K	K	4	4	¬	♭NOT♭
L	L	5	5	&	♭AND♭
M	M	6	6	|	♭OR♭
N	N	7	7	<	♭LT♭
O	O	8	8	>	♭GT♭
P	P	9	9	_	unavailable
Q	Q	♭(blank)	♭	?	unavailable
R	R	=	=	@	unavailable
S	S	+	+	=	unavailable
T	T	–	–	$	$

restricted set (see Table 18.2). PL/I is informed as to which set is being used by special instructions to the operating system outside the scope of the program itself.

The 60-character set is divided into 10 *numeric* (0–9), 29 *"alphabetic"* characters (A–Z, $, #, and @) and 21 *special characters*. The *break character* (_) is usually used to make names more readable. For instance a variable called MSFLOW in FORTRAN and MASS-FLOW in COBOL might be called MASS_FLOW in PL/I.

18.1.2 PL/I's Vocabulary

As is true in any other language, PL/I has a certain repertoire of *keywords* which, in certain contexts, trigger particular compiler operations. In contrast to other languages, however, these words are not reserved. They

TABLE 18.2 PL/I OPERATIONAL SYMBOLS

Operation	60-character Symbol	48-character Symbol
Greater than or Equal To	>=	♭GE♭
Less than or Equal To	<=	♭LE♭
Not Equal To	¬=	♭NE♭
Concatenation	||	♭CAT♭
Pointer Qualifier	–>	♭PT♭

may be used as variable names, labels, and so forth, as long as the programmer is willing to keep his own books straight. The compiler will process such names consistently. For example, the statement

```
DECLARE X;
```

will reserve storage for a variable named X. By the same token,

```
DECLARE DECLARE;
```

will reserve storage for a variable named DECLARE. As long as you know what is going on, everything is fine. Certain keywords have acceptable abbreviations, as we shall see later.

18.1.3 Basic Coding Format

When necessary we shall orient our discussion to the punched card as our program input vehicle. The basic PL/I instruction is called a *statement.* Although many different statement types may be written, they must all end with a semicolon. Statements may be labeled or unlabeled. When a label is used it precedes the statement and ends with a colon. A label must begin with one of the 29 alphabetic characters and may contain alphabetics, numerics or break characters (no blanks). Its length cannot exceed 31 characters. More than one label may be attached to a statement:

*label*1:*label*2: . . . :*statement;*

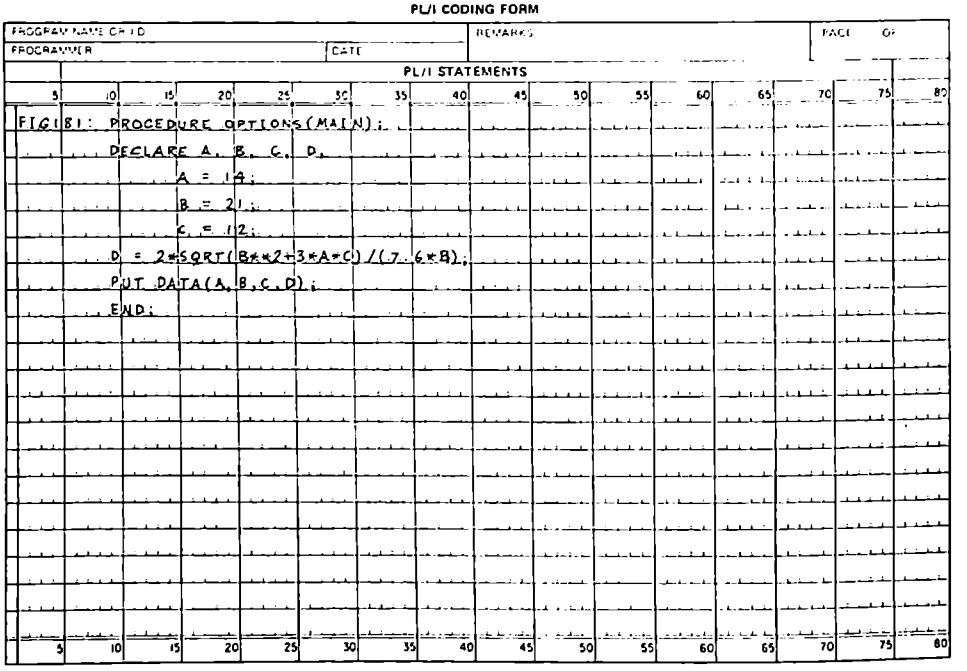


Figure 18.1 PL/I coding form

Since the colon serves as punctuation, the use of blanks on either side of it is optional, and PL/I ignores them. The same is true in the presence of the semicolon, parenthesis, quote mark, plus sign or other symbol that explicitly separates two statement components.

Most installations are set up so that column 1 of each card is off limits for PL/I program statements (but *not* for data). Other than that there are no format restrictions. Statements may begin anywhere on the card and need not end on that card. The only rule is that if a word (keyword or otherwise) is broken at column 80, it must be continued in column 2 of the next card, so that there are no intervening blanks (remember that column 1 is out).

Comments may appear anywhere that blanks may be inserted. Each comment is preceded by /* and followed by */. These appear in the source program listing but are not processed. A sample PL/I program is shown in Figure 18.1.

18.1.4 Program Organization

A PL/I program consists of one or more *procedures.* One of these (usually the first) must be a *main procedure* and the others, either contained in it or attached to it, are subordinate. Each procedure begins with a PROCEDURE statement and concludes with an END statement. For the main procedure, the initial statement takes the form

procname: PROCEDURE OPTIONS (MAIN);

(PROCEDURE may be abbreviated PROC). The END statement may or may not refer to the procedure name (though it is a good idea to do so):

END *procname;*

END has several other uses, so that it is quite common to have several such statements in a single procedure.

18.2 SPECIFICATIONS FOR DATA

PL/I recognizes and handles a wide variety of data types. Each of these may be specified in a number of ways without direct concern about conversion. The compiler contains an extensive array of conversion mechanisms, many of which are actuated as part of the default operations. Although program-control information can be considered as data, we shall restrict our present discussion to problem data.

18.2.1 Constants

It is convenient to divide problem data into *arithmetic* and *string* types. This categorization is consistent for PL/I, since a different set of operations (and operators) is associated with each category.

TABLE 18.3 FORMS FOR PL/I NUMERIC CONSTANTS

Conventional Notation	Base	Scale	Mode	PL/I Form
72.1	DECIMAL	FIXED	REAL	72.1
.0341				.0341
-158				-158
$72.1 + 3i$			COMPLEX	72.1 + 3I
$.0341 - .65i$				.0341 − .65I
$-18.87i$				0 − 18.87I
$.6247 \times 10^{-4}$		FLOAT	REAL	.6247E−4
$-.0412 \times 10^{2}$				−.0412E2
-65.7×10^{-6}				−65.7E−06
14×10^{-2}			COMPLEX	14E−2 + 0I
$6.6 \times 10^{1} - .0372i$				6.6E+4 − .0372E0I
$.7041 \times 10^{-1} + 1.53 \times 10^{2}i$				.7041E−1 + 1.53E02I
11011_2	BINARY	FIXED	REAL	11011B
101.0101_2				101.0101B
$.001101_2$				.001101B
$1_2 + 1.01_2i$			COMPLEX	1B + 1.01BI
$.011_2 - 101.001_2i$				.011B − 101.001BI
$.01011_2i$				0B + .01011BI
$1.01_2 \times 2^{101_2}$		FLOAT	REAL	1.01E+5B
$-.0011_2 \times 2^{-11_2}$				−.0011E−3B
$100_2 \times 2^{111_2}$				100E7B
$-.011_2 \times 2^{-10_2} + 10.1_2 \times 2^{-11_2}i$			COMPLEX	−.011E−2B + 10.1E3BI
$1.1_2 \times 2 - .1_2 \times 2^{100_2}i$				1.1E1B − .1E4BI
$101.1_2 \times 2^{-100_2}i$				0B + 101.1E−6BI

18.2.1.1
Arithmetic Constants

Arithmetic data are characterized by four attributes:

(a) Base (DECIMAL (DEC) or BINARY (BIN))
(b) Scale (FIXED or FLOAT)
(c) Mode (REAL or COMPLEX (CPLX))
(d) Precision (coded description)

These are either specified explicitly when the programmer reserves space for them in DECLARE statements, or they may be assigned by default. The forms for expressing numeric constants in PL/I generally resemble conventional usage, so that most attributes of a constant can be inferred from its appearance. The various forms are summarized in Table 18.3.

Although floating-point numbers are ultimately stored using a hexadecimal base (Chapter 2), PL/I handles this as a special case of binary representation.

PL/I provides facilities for handling arithmetic with English currency. In support of this, the programmer may specify *Sterling constants* using the general form

*a.b.c.d*L

TABLE 18.4 EXAMPLES OF STRING CONSTANTS

Character Strings		Bit Strings	
Values to be Stored	PL/I Form	Values to be Stored	PL/I Form
3J2.7	'3J2.7'	010100	'010100'B
16.42	'16.42'	0000	'0000'B
FRUMP	'FRUMP'	1010	'1010'B
CAN'T	'CAN''T'	02011	can't do it
'TWASN'T	''TWASN''T'	11J0S	can't do it
1001101	'1001101'	1001101	'1001101'B

where a, b, and c are the number of pounds, shillings and pence, respectively, and d is fractional pence.

18.2.1.2 String Constants

These constants are either CHARACTER strings or BIT strings. The latter may contain only 1's and 0's and are treated as sequences of individual bits rather than as binary numbers. All string constants must be enclosed in single quote marks. Some examples are given in Table 18.4.

18.2.2 Declaration of Variables

The DECLARE statement (abbreviated DCL) is PL/I's basic vehicle for allocating storage for and assigning names and attributes to variables. Its basic form is

DECLARE *v*1 *attr*1, *attr*2, . . ., *v*2 *attr*1', *attr*2', . . .;

*v*1 and *v*2 are the programmer-designated names for variables, each of which is followed by its list of attributes. As indicated, a number of variables may be declared in a single statement.

Numerical variables are declared with appropriate base, scale and mode keywords in any order. Thus the following two declarations

```
DECLARE X REAL FIXED DECIMAL;
DECLARE X FIXED DECIMAL REAL;
```

are exactly equivalent.

The precision of a numerical variable is specified after the base, scale and mode attributes. For FIXED variables (regardless of base or mode), the precision is listed in the form

(m,n)

where m is the total number of digits and n specifies how many of these m digits are to be considered as being to the right of the decimal (or binary) point. Thus,

```
DECLARE Y FIXED REAL DECIMAL (5,2);
```

reserves sufficient storage under the name Y to accommodate a number in the form XXX.XX. If *n* is zero it, and the preceding comma, may be omitted. For FLOAT variables, the precision is defined merely as (*n*). For instance,

```
DECLARE C FLOAT REAL DECIMAL (6);
```

will reserve sufficient storage under the name C to accommodate a value in the form 0.XXXXXXE±XX.

String variables are declared with either the CHARACTER (abbreviated CHAR) or BIT attribute, followed by a parenthesized integer indicating the number of characters or bits to be allocated. Thus,

```
DECLARE E CHARACTER (4), G BIT (7);
```

will allocate four bytes under the name E and seven bits under the name G.

18.2.2.1 The INITIAL Option

A starting value may be assigned to a variable as part of its declaration by following its attribute list with the phrase

```
INITIAL (value)
```

(INITIAL may be abbreviated INIT). For numerical variables, the specified constant need not have the same attributes as listed in the declaration. PL/I will make the necessary conversion. For example, the statement

```
DECLARE X FLOAT REAL DECIMAL (7) INITIAL (27.4), B
CHARACTER (7) INITIAL ('DIDN''T');
```

will result in the storage of the value 0.2740000E+02 in X and the seven characters DIDN'T␣ in B.

18.2.2.2 Factoring of Declarations

When several variables are to be declared with a common set of attributes, the DECLARE statement may be shortened by using the *factored* form. For instance, the statement

```
DCL X FIXED REAL DECIMAL (4,1), Y FIXED REAL DECIMAL (4,1);
```

can be written

```
DCL (X,Y) FIXED REAL DECIMAL (4,1);
```

and the statement

```
DCL A FLOAT REAL DECIMAL (8), INITIAL (0), B FLOAT REAL
BINARY (3) INITIAL (0);
```

can be shortened to

```
DCL (A DECIMAL, B BINARY) REAL FLOAT INITIAL (0);
```

18.2.2.3 The VARYING Attribute

When a string variable is declared, the assigned number of bytes (or bits) is fixed. Even when a particular value requires less storage, it is padded to the right with blanks (or zeros for bit strings) to fill the entire length. By adding the keyword VARYING (abbreviated VAR) to the declaration, it is possible to allocate a given amount of storage which will expand or shrink as required. Thus

```
DCL D CHARACTER (9) VARYING;
```

makes up to 9 bytes available under the name D.

18.2.2.4 An Alternate Description Method—the PICTURE Attribute

Declared variables may be described on a character-by-character basis using the PICTURE (abbreviated PIC) attribute. This is a direct carryover from COBOL (see Chapters 15–17) and is provided as an alternative specification method. Instead of listing the various attributes, this type of description consists of

```
DCL vI PICTURE 'codes';
```

For fixed-decimal variables, each digit is indicated by a 9 and the decimal point's position is designated by a v. For instance,

```
DCL A REAL FIXED DECIMAL (6,2), B PICTURE '9999v99';
```

will cause PL/I to treat A's and B's values in the form XXXX.XX. The same type of description is given for fixed binary variables, except that 1's are used instead of 9's.

Floating-point variables are described in the same general way, with the addition of a K to indicate the beginning of the exponent portion. To illustrate,

```
DCL X FLOAT BINARY REAL (5);
```

and

```
DCL X PICTURE 'v11111K99';
```

are functionally equivalent.

String variables are specified with combinations of three codes:

1) 9—the character in this position can only be a number.
2) A—the character in this position may only be one of the 29 alphabetics or blank.
3) X—any character may appear in this position.

Thus

```
DCL G PICTURE 'AXX99';
```

allocates five bytes for a variable named G whose first character must be alphabetic, or blank, the next two may be anything, the last two must be numbers. We shall have more to say about this later.

18.2.2.5
Naming PL/I Variables

Variables are named according to the same rules defined for statement labels (Sec. 18.1.3). There is no restriction on the use of specific types of names for variables with particular attributes. However, there are certain default mechanisms which operate in the absence of explicit specifications, as we shall see next.

18.2.3
Default Mechanisms in Declarations

It is not always necessary to include specifications for all attributes when declaring a variable in PL/I. There is a complete set of default mechanisms which will automatically make assignments using prescribed values.

If no attributes at all are given, PL/I assumes that the named variable is a REAL number. Base, scale, and precision are then assigned depending on the name's first letter. For names beginning with I, J, K, L, M, or N, the attributes FIXED and BINARY are assigned, together with a precision of (15,0). When other initial letters are used, the default attributes are DECIMAL FLOAT (6). It should be noted that the default precision may vary with each installation; the values given here are typical rather than fixed. This name convention is easily overridden by specifying a base and/or scale, as shown in Table 18.5.

TABLE 18.5 DEFAULT ATTRIBUTE ASSIGNMENTS (REGARDLESS OF NAME)

If Programmer Specifies	PL/I Assumes
Either Base & Either Scale	REAL
Either Base & Either Mode	FLOAT
Either Scale & Either Mode	DECIMAL
Either Mode	DECIMAL, FLOAT
Either Scale	REAL, DECIMAL
Either Base	REAL, FLOAT

For string variables (in which CHARACTER or BIT must always appear, of course) the default length is 1.

18.3
ARITHMETIC ASSIGNMENT STATEMENTS

Arithmetic assignment statements in PL/I closely follow the construction used in FORTRAN (Chapters 12–14). The general form is

v 1 = *expression;*

where *expression* may be a single constant, a single variable or a combination of these terms connected by arithmetic operators. The = sign in this context is the replacement operator, so that the statement

```
X = X + 5;
```

is read "add 5 to the current value in X and place the result back in X." An arithmetic expression may contain mixed scales, bases, modes, and precisions. PL/I will make the necessary conversions according to a prescribed set of priorities.

An extended form of the assignment statement allows the programmer to write statements such as

```
X,Y,Z = X + 5;
```

in which case 5 will be added to X's current contents, with the result being placed in Y and Z as well as X.

EXAMPLE 90

Write a program that sets A and B to 8.38 and 16.35, respectively, and calculates C as $\sqrt{A^2 + B^2}) \, AB$, storing the result as XX.XXXX.

Solution Program:

```
EX90:  PROCEDURE OPTIONS (MAIN);
       DECLARE (A,B) FIXED (4,2), C FIXED (6,4);
       A = 8.38; B = 16.35;
       C = (A**2 + B**2)**.5/(A*B);
       END EX90;
```

18.4 BUILT-IN FUNCTIONS

PL/I has a large repertoire of routines which perform a variety of specific calculations. These routines, called *built-in functions,* are activated by their respective keywords, which may be used in arithmetic expressions. When a built-in function is used, its specific keyword is followed by a parenthesized list of one or more *arguments,* the number and type being set for each function as part of its requirements.

The design of these functions is such that there is usually no restriction on the base, scale, or precision of their arguments. For example, the keyword LOG10 activates a function which calculates the logarithm (to base 10) of a single numeric argument. The only requirement is that the argument be a positive number. It happens that the actual logarithm is computed using a floating-point argument. However, this is of no direct concern to the programmer, since PL/I will handle any required preconversion. Thus in the general assignment statement

```
Y1 = LOG10(X);
```

the argument x may be anything which yields a single number greater than zero. This includes a constant, variable, or a single-valued arithmetic expression of any complexity. Furthermore, once the logarithm is obtained it will automatically be expressed in the form dictated by the declared and implied attributes of the destination variable v1. Because of this versatility, these functions are called *generic* functions.

EXAMPLE 91

Repeat Example 90 using the built-in square root function (keyword is SQRT).

Solution Program

```
EX91:  PROCEDURE OPTIONS (MAIN);
       DECLARE (A,B) FIXED (4,2), C FIXED (6,4);
       A = 8.38; B = 16.35;
       C = SQRT (A**2 + B**2)/(A*B);
       END EX91;
```

Table 18.6 summarizes PL/I's arithmetic built-in functions.

TABLE 18.6 ARITHMETIC BUILT-IN FUNCTIONS

X and *Y* are real arguments; *n* and *m* are decimal integers, *C* and *E* are complex arguments. *D* is an angle in degrees, *R* is an angle in radians.

Keyword and Usage	Result
ABS(*X*)	$\|X\|$
SIGN(*X*)	SIGN (*X*) is -1, 0 or 1 when *X* is negative, zero or positive, respectively.
PRECISION(*X,n,m**)	*X*'s precision is set to (*n,m*) or (*n*) if *m* is not specified.
FIXED(*X,n,m**)	*X* is converted to FIXED with precision (*n,m*).
FLOAT(*X,n**)	*X* is converted to FLOAT with precision (*n*).
BINARY(*X,n*m**)	*X* is converted to BINARY with precision (*n,m*).
DECIMAL(*X,n*,m**)	*X* is converted to DECIMAL with precision (*n,m*).
MOD(*X,Y*)	*X* is divided by *Y* and the *remainder* is expressed as an integer; e.g., MOD(17,5) is 3.
MAX(*X,Y,etc*)	The largest value in the list is selected. (Not which one; just how much.)
MIN(*X,Y,etc*)	The smallest value in the list is selected. (Not which one; just how much.)
FLOOR(*X*)	The largest integer $\leq X$.
CEIL(*X*)	The smallest integer $\geq X$.
TRUNC(*X*)	Same as FLOOR(X) for $X \geq 0$, same as CEIL(X) for $X < 0$.
ADD(*X,Y,n,m**)	$X + Y$ with precision (*n,m*) or (*n*).
MULTIPLY(*X,Y,n,m**)	$X * Y$ with precision (*n,m*) or (*n*).
DIVIDE(*X,Y,n,m**)	X/Y with precision (*n,m*) or (*n*).
REAL(*C*)	If *C* is $a = bi$ REAL(*C*) is *a*.
COMPLEX(*X,Y*)	$X + Yi$
CONJG(*C*)	If *C* is $r + si$, CONJG(*C*) is $r - si$.
EXP(*X*)	e^n
LOG(*X*)	$\log_e X$
LOG10(*X*)	$\log_{10} X$

TABLE 18.6 (cont'd)

Keyword and Usage	Result
LOG2(X)	$\log_2 X$
ROUND(X,n)	X is rounded to n digits to the right of the decimal point.
ATAND(X)	$\tan^{-1} X$, expressed in degrees.
ATAN(X)	$\tan^{-1} X$, expressed in radians
TAND(D)	Tangent of D
TAN(R)	Tangent of R
SIND(D)	Sine of D
SIN(R)	Sine of R
COSD(D)	Cosine of D
COS(R)	Cosine of R
TANH(R)	tanh R
SINH(R)	sinh R
COSH(R)	cosh R
STANH(X)	$\tanh^{-1} X$
ERF(X)	$\frac{2}{\sqrt{\pi}} \int_0^X e^{-X^2}\, dX$
ERFC(X)	1 − ERF(X)
SQRT(X)	$\sqrt{X}$
ATAN(Y,X)	See nested table below
ATAND(Y,X)	(180/π) * ATAN(Y,X)

Value of Arguments	Result, Radians
$X > 0$	$\tan^{-1}\frac{Y}{X}$
$X = 0,\ Y > 0$	$\pi/2$
$X, Y = 0$	Error
$X = 0,\ Y < 0$	$-\pi/2$
$X < 0,\ Y > 0$	$\pi + \tan^{-1}\frac{Y}{X}$
$X < 0,\ Y < 0$	$-\pi = + \tan^{-1}\frac{Y}{X}$

*optional

18.5 DATA AGGREGATES

PL/I provides extensive facilities for defining and manipulating two basic types of data collections. Arrays are direct carryovers from FORTRAN while structures are lifted from COBOL.

18.5.1 Arrays

Arrays are declared like single-valued variables, with the number of elements in each dimension given in parentheses immediately after the name. Thus,

```
DECLARE TABLE (6,15) FIXED (2);
```

reserves sufficient storage for a collection of 90 two-digit real decimal fixed-point integers named TABLE and organized into six rows and fifteen columns. There is no practical limit on the number of dimensions that may be specified, and array declarations may be factored along with other arrays and/or single valued variables.

An individual element of an array is referred to by the array name followed by a set of parenthesized subscripts representing row number, column number, etc. A subscript, like an arithmetic function argument, may be any legitimate single-valued arithmetic expression. PL/I will evaluate it and truncate to an integer if it does not come out that way.

PL/I arrays are stored internally in *row major order*. That is, the last subscript varies the fastest. Thus an array declared as A(2,4,3) is stored in the order A(1,1,1),A(1,1,2),A(1,1,3),A(1,2,1),A(1,2,2),A(1,2,3), A(1,4,2),A(1,4,3),A(2,1,1), . . . A(2,4,1),A(2,4,2),A(2,4,3).

18.5.1.1 Handling Entire Arrays with Single Statements

We can process an entire array with a single assignment statement, as long as certain rules are followed and understood. To see how this works, let us make the following illustrative declarations:

```
DECLARE (A(3), B(3), C(3), D(2,3), E(3,2), F(6), X, Y) FIXED(3);
```

Now we can see what happens. The simple assignment,

```
A = X + 7;
```

is exactly the same as

```
A(1) = X + 7;
A(2) = X + 7;
A(3) = X + 7;
```

We can involve several arrays as long as they are dimensioned *exactly* the same. For instance,

```
C = A*B + 2/Y;
```

is equivalent to

```
C(1) = A(1)*B(1) + 2/Y;
C(2) = A(2)*B(2) + 2/Y;
C(3) = A(3)*B(3) + 2/Y;
```

Note, however, that we cannot write

```
F = 2 + D; or D = E;
```

because, although F, D, and E each have the same number of elements, and D and E have the same number of dimensions, there is no complete correspondence.

Another thing must be watched. Since an array element is just like any other single-valued variable, it may be used as such in expressions, as long as care is taken. To illustrate, we shall assign the following values to D:

row 1	104	200	68
row 2	650	511	124

Now if we were to write

```
D = D/D(1,2);
```

expecting to divide each element of D by 200, we shall be disappointed. The sequence will be

```
D(1,1) = D(1,1)/D(1,2):  giving a D(1,1) of 104/200
D(1,2) = D(1,2)/D(1,2):  giving a D(1,2) of 200/200 or 1.
```

Now, since D(1,2) has changed, we shall be dividing the rest of D's elements by 1 rather than by 200.

18.5.1.2 Array Cross Sections

It is possible to refer collectively to parts of arrays by using asterisks in the subscripts. For example, using D declared in the previous section, D(1,*) refers to all elements in row 1, D(1,1), D(1,2), and D(1,3). This is known as a *cross section*. Once such a cross section is identified, it may be handled as if it were a complete array. Thus, D(1,*) will act as a one dimensional array with three elements and we can write something like

```
D(1,*) = 2*SQRT(B) + A;
```

(using A and B from the previous section). This is equivalent to stating

```
D(1,1) = 2*SQRT(B(1)) + A(1):
D(1,2) = 2*SQRT(B(2)) + A(2):
D(1,3) = 2*SQRT(B(3)) + A(3);
```

EXAMPLE 92

G is a 4 × 4 array of two-digit integers. Assuming that its elements' values are assigned elsewhere in the program, write the statements to create an array TG which is the transpose of G (rows interchanged with columns).

Solution Program

```
EX92:  PROCEDURE OPTIONS (MAIN);
DECLARE (G(4,4),TG(4,4))FIXED(2);

.
/* VALUES ASSIGNED SOMEWHERE IN HERE.*/
```

Solution Program (continued)

```
TG(1,*) = G(*,1);
TG(2,*) = G(*,2);
TG(3,*) = G(*,3);
TG(4,*) = G(*,4);
END EX92;
```

18.5.1.3
Special Subscripts

The declaration

```
DECLARE Z(6,4) FIXED(3);
```

implies that Z's row numbers will vary from 1 to 6 and its column numbers from 1 to 4. PL/I allows the programmer to specify whatever range he wishes. This range may include subscript values of zero or less. The form is shown in the example below. If we said

```
DECLARE Z(-3:2,6:9)FIXED(3);
```

PL/I would still reserve the same amount of storage as before (for a 6×4 array), but the first element would be called Z(−3,6) instead of Z(1,1) and the final element would be Z(2,9). If we were to write

```
DECLARE Z (-3:2,4)FIXED(3);
```

Z's rows would be numbered −3, −2, −1, 0, 1, 2 and its columns would be 1, 2, 3, and 4.

EXAMPLE 93

A table of density values for a given material is to be stored in memory for reference. This table consists of values for temperatures ranging from −50°F to + 100°F in one-degree increments. Write a series of statements which take a temperature value TEMP and a volume VOL and, using the density at that temperature, produce a value of WT as the product of volume and density. Assume precision of (4,3) for densities, (4,1) for volume, and (8,4) for weight.

Solution Program

```
EX93: PROCEDURE OPTIONS (MAIN);
      DECLARE DNSTY (-50:100)FIXED(4,3),
        TEMP FIXED(4,3), VOL FIXED(4,1),WT FIXED(8,4);

      .
      /* SOMEHOW WE GET VALUES FOR DNSTY,VOL,TEMP. */
```

```
    WT = VOL*DNSTY(TEMP);
END EX93;
```

18.5.1.4 Built-in Functions for Arrays

The arithmetic generic functions described earlier may be used with arrays or cross section as arguments. Thus if A and B are each three-element one-dimensional arrays, for example, the statement

```
A = 2*SQRT(B);
```

is perfectly fine. In addition, there are several functions specifically designed for array arguments.

1. The SUM function will compute the sum of the elements in an array expression. Using the example names in 18.5.1.1, the statement

   ```
   Y = 2*SUM(A+B);
   ```

   will store the single value 2*((A(1)+B(1)) + (A(2)+B(2)) + (A(3)+B(3)) in Y.
2. The PROD function will compute products, so that

   ```
   Y = .5*PROD(A+B);
   ```

   places the value .5*((A(1)+B(1))*(A(2)+B(2))*(A(3)+B(3))) in Y.
3. The POLY function evaluates polynomials (in its simplest version). The basic form is

   ```
   POLY (a1,a2)
   ```

   where *a*1 is a one-dimensional array (or a single vector (like one row) from a multidimensional array) and *a*2 is a single-valued argument. The single value produced is

   $$a1_1 + a1_2a2 + a1_3a_2^2 + a1_4a_2^3 + \cdots$$

18.5.2 Structures

In addition to arrays, in which all elements must have a common set of attributes, PL/I provides a mechanism for the manipulation of diverse collections of data arranged in hierarchical organizations. These are called *structures*. A structure is named and described in a DECLARE statement starting with the overall name for the collection and working down into the various subordinate components. The hierarchical relationship among components is indicated by *level numbers*, with the overall name receiving a level of 1 (the only level 1 in a structure) and the various subsidiaries receiving numerically higher levels. A structure (or *major* structure) may

contain within it other (minor) structures. Whenever a name is used as a collective reference to several items in a structure, it should be understood that this name is not declared with any attributes other than a level number. No storage is actually associated with the collective name. Thus if we said

```
DECLARE 1 I_D,
  2 NAME,
    3 LAST CHARACTER(18),
    3 LETTER CHARACTER(1),
  2 AGE FIXED(2),
  2 HEIGHT FIXED(2),
  2 WEIGHT FIXED(3);
```

I_D is the major structure consisting of a two-digit integer fixed-point variable called AGE and HEIGHT, a three-digit integer variable called WEIGHT and a minor structure named NAME, consisting in turn of an 18-character field called LAST and a single-character variable called LETTER. Structures may contain arrays as members and arrays of major structures may be declared.

When several structures are declared with identical hierarchical organization, they may be operated on with single statements, as we have done with arrays. This is also true for minor structures with identical organizations, even if they appear in different-looking major structures.

EXAMPLE 94

A payroll record, previously stored in memory, contains three fields: name, hours, and rate. The hours field is subdivided into regular hours and overtime hours subfields. Write a program to compute regular pay, overtime pay (at one and one-half the regular rate), and total pay. Then form an output record consisting of name and pay, with pay being subdivided into regular, overtime, and total pay.

Solution Program

```
GROSS_PAY: PROCEDURE OPTIONS (MAIN);
  DECLARE 1  PAYROLL,
    2 NAME,
      3 FIRST CHARACTER (12),
      3 LAST CHARACTER (8),
      3 MIDDLE CHARACTER (1),
    2 HOURS,
      3 REGULAR PICTURE '99v99',
      3 OVERTIME PICTURE '99v99',
    2 RATE PICTURE '9v99',
    1 OUTPUT_RECORD,
    2 NAME,
      3 LAST CHARACTER (12),
      3 FIRST CHARACTER (8),
```

```
    3 MIDDLE CHARACTER (1),
  2 PAY,
    3 REGULAR PICTURE '999v99',
    3 OVERTIME PICTURE '999v99',
    3 TOTAL PICTURE '999v99';

/* VALUES ARE STORED SOMEHOW. */

OUTPUT_RECORD.NAME = PAYROLL.NAME;
PAY. REGULAR = RATE * HOURS.REGULAR;
PAY. OVERTIME = 1.5 * RATE * HOURS.OVERTIME;
TOTAL = PAY.REGULAR + PAY.OVERTIME;
END GROSS_PAY;
```

Note that we used PL/I's format flexibility in our declaration. It would have been just as valid (though harder to follow) if we had written

```
DECLARE 1 PAYROLL, 2 NAME, 3 LAST CHARACTER
(12), 3 FIRST CHARACTER (8), 3 MIDDLE etc.
```

The PICTURE descriptions were thrown in as a reminder that this capability is also available.

Five fields of PAYROLL have the same names as fields of OUTPUT_RECORD. By qualifying identical names, there is no ambiguity. OUTPUT_RECORD.NAME refers to the NAME structure of the OUTPUT_RECORD structure; PAYROLL.NAME refers to the NAME structure of the PAYROLL structure, PAY.REGULAR refers to the REGULAR field of the PAY structure, and so forth.

18.6 CHARACTER AND BIT STRING HANDLING

PL/I has a number of character and bit-handling facilities built into its basic design which provide capabilities paralleling those for numeric data.

18.6.1 Character Manipulations

The basic character-handling operation is *concatenation*, symbolized by ||. This consists merely of tying character strings together to form longer ones. For example, if we had the sequence

```
DECLARE A CHARACTER (5) INITIAL ('GRONK').
        B CHARACTER (7);
B = 'NA'||A;
```

we would be placing the seven characters NAGRONK in B. Character strings may be built by concatenating as many terms as are required.

18.6.1.1 Replacement with Unequal-Length Strings

When an assignment statement specifies a destination that is shorter than what is supposed to go into it, truncation occurs on the right. Thus if we said

```
DECLARE B CHARACTER (5), C CHARACTER (7);
C = 'AUSPETL'; B = 'FRIMZ';
B = C;
```

B would end up with AUSPE. When the opposite is true, the surplus characters are filled in with blanks; they are *not* left with their previous contents. Using B and C from before,

```
C = B;
```

will place FRIMZ␣␣ in C, not FRIMZTL. Assignment statements may be used for entire arrays as with numbers.

18.6.1.2 Built-in Functions for Character Strings

The concatenation operation is supplemented by several powerful character-handling functions:

1. The LENGTH function permits the programmer to obtain the current length of a string declared with the VARYING attribute. Thus

   ```
   Y = LENGTH (C);
   ```

   places an integer in Y corresponding to C's current length.
2. The SUBSTR function is a very versatile routine that provides capabilities for handling parts of strings independently. Its basic form is

   ```
   SUBSTR (v1,a2,a3)
   ```

   where *v*1 is the name of the character string in which the desired part (substring) is to be found; *a*2 is a numeric variable or expression indicating the position at which the substring is to begin; *a*3, whose inclusion is optional, is a variable or expression indicating the length of the substring. If *a*3 is omitted, the substring will go from position *a*2 of *v*1 till the end. For example, if A contains TRINKET and B is declared with length 7,

   ```
   B = SUBSTR (A,3,3);
   ```

   will place INK␣␣␣␣ in B, while the statement

   ```
   B = 'PL' ||SUBSTR(A,3);
   ```

   will place PLINKET in B.

Substrings may appear on the left-hand side of an assignment statement. This is not true for other functions, arithmetic or otherwise. When used this way, the expression is called a *pseudovariable.* If A contains the character string TRINKET, the statement

```
SUBSTR (A,3,3) = 'UMP';
```

changes the third, fourth and fifth characters in A, resulting in TRUMPET.

3. The INDEX function is used to determine whether a particular combination of characters is a substring of a specified variable. Its form is

   ```
   INDEX (v1,v2)
   ```

   and the result is an integer indicating the position in *v*1 at which a string identical to *v*2 begins. If there is none, a zero is given. For example, if A contains TRINKET, then

   ```
   Y = INDEX (A,'NKE');
   ```

   will place a 4 in Y. Using the same A and a string B containing RETRIEVE, the statement

   ```
   Y = INDEX (A,SUBSTR(B,3,3));
   ```

   is, in effect, an instruction to look for the string TRI in A. Hence, the function will place a 1 in Y.
4. The STRING function will concatenate an entire array of strings to form a single one. For instance, if we say

   ```
   DECLARE A(2,3) CHARACTER (2), B CHARACTER (12);
   ```

   then the following two statements are equivalent;

   ```
   B = STRING (A);
   B = A(1,1)||A(1,2)||A(1,3)||A(2,1)||A(2,2)||A(2,3);
   ```

18.6.2 Bit Strings

Although bit strings consist of nothing but 1's and 0's, these are not treated as numbers. Instead, they are handled as multiple switches and are manipulated by a separate set of Boolean operators.

18.6.2.1 Logical Operations

PL/I provides the three basic logical operators AND (&), OR (|), and NOT (¬). These may be used to form logical expressions which, irrespective of their length, work down to either *true* ('1'B) or *false* ('0'B). To illustrate their use, we shall first restrict ourselves to bit strings of length

TABLE 18.7 LOGICAL OPERATIONS

X	Y	¬Y	X&Y	X|Y	X&¬Y	X|¬Y
'0'B	'0'B	'1'B	'0'B	'0'B	'0'B	'1'B
'0'B	'1'B	'0'B	'0'B	'1'B	'0'B	'0'B
'1'B	'0'B	'1'B	'0'B	'1'B	'1'B	'1'B
'1'B	'1'B	'0'B	'1'B	'1'B	'0'B	'1'B

1. The results of logical operations are usually expressed in *truth tables*. A simple summary of &, |, and ¬ is given in this form in Table 18.7.

When used with longer bit strings, a bit-by-bit comparison is made, producing a result dictated by the type of expression. If the operands are of unequal lengths, PL/I adds zeros on the right to bring all the operands up to the longest one.

As, an example, let us evaluate the statement

C = A &¬(B|D);

for A = '1011'B, B = '1000'B, and D = '0100'B. First we set up our truth table:

A	*B*	*D*	*B*|*D*	¬(*B*|*D*)	*C* = *A*&¬(*B*|*D*)
'0'B	'0'B	'0'B	'0'B	'1'B	'0'B
'0'B	'0'B	'1'B	'1'B	'0'B	'0'B
'0'B	'1'B	'0'B	'1'B	'0'B	'0'B
'0'B	'1'B	'1'B	'1'B	'0'B	'0'B
'1'B	'0'B	'0'B	'0'B	'1'B	'1'B
'1'B	'0'B	'1'B	'1'B	'0'B	'0'B
'1'B	'1'B	'0'B	'1'B	'0'B	'0'B
'1'B	'1'B	'1'B	'1'B	'0'B	'0'B

Now applying these criteria to our values, we obtain

A	1	0	1	0
B	1	0	0	0
D	0	1	0	0
C	0	0	1	1

so that under these conditions '0011'B is placed in C.

18.6.2.2 Built-in Functions for Bit Strings

In addition to the character handling functions (Sec. 18.6.1.2) there are several special routines for bit strings:

TABLE 18.8 PROPERTIES OF THE BOOL FUNCTION

Value of a Bit in *v1*	Value of the Corresponding Bit in *v2*	Outcome Value Specified by the Bit in
'0'B	'0'B	position 1 of *a3*
'0'B	'1'B	position 2 of *a3*
'1'B	'0'B	position 3 of *a3*
'1'B	'1'B	position 4 of *a3*

1. The BOOL function allows the programmer to set up his own logical operations by specifying any desired set of outcome rules. This is done by writing

   BOOL (*v1*,*v2*,*v3*)

   where *v1* and *v2* are the two bit strings to be compared. *v3* is a four-bit string or the name of a four-bit string which symbolizes the rules to be used for *v1* or *v2*. Table 18.8 shows how this works.
2. UNSPEC expresses each character as the equivalent string of eight bits. For example, if B contains the two characters J3, and C is declared as BIT(16), the statement

   C = UNSPEC(B);

   will place 1101000111110011 in C, since J is 11010001 and 3 is 11110011 in EBCDIC (see Chapter 2).
3. The ALL function is intended for use on arrays of bit strings. In the statement

   Y = ALL(A);

   A is an array of bit strings and Y is a single string having the same length as each of A's members, then the ALL function will place a 1 in a given position in Y if that position contains a 1 for all elements of A. Otherwise, a zero will be placed there.
4. The ANY function is somewhat contrary to ALL. It will place a one in a position of string Y if any of array A's members has a 1 in that position.

18.7 LOOPS AND DECISION STATEMENTS

PL/I provides what is probably the most extensive repertoire of decision capabilities available in a programming language. In general, the pertinent statements are fairly straightforward, thus facilitating the synthesis of decision networks.

18.7.1 Sequence Changing-the GO TO Statement

Unconditional branching is accomplished with the simple statement

GO TO *label;*

where *label* is some statement label used in the procedure. A more flexible version of the GO TO may be specified by using *label variables*. That is, PL/I allows the declaration of a name with the LABEL attribute, thereby providing a variable which may contain nothing but a statement label. This is *not* a character string and cannot be set equal to a character string. Also, the label variable itself cannot be attached to a statement. Thus if there is some label HERE attached to a statement and we say

```
DECLARE WHERE LABEL;

.
WHERE = HERE;


GO TO WHERE;
```

the program will branch to the statement labeled HERE. Then at some other point we set WHERE to some other value, the same GO TO statement will now have another destination.

In addition to the basic declaration shown above, there are three other forms:

1. DECLARE START LABEL (LABEL1, LABEL2, LABEL3); specifies that START may assume as a value any of the statement labels in the explicit list.
2. DECLARE START LABEL (LABEL1, LABEL2) INITIAL (LABEL2); specifies that START may assume as a value any of the statement labels in the explicit list and, in addition, is assigned initially the statement label given with the INITIAL keyword.
3. DECLARE START (3) LABEL INITIAL (LABEL1, LABEL2, LABEL3); specifies that START has three values, initially as given with the INITIAL keyword, and that they may be referenced by subscripting. When I = 2, for example, START (I) assumes as a value the second of its three statement labels.

18.7.2 The IF Statement

In order to handle decisions in PL/I with any facility, it is necessary to understand the mechanism behind the *relational operators*. There are six such operators:

Symbol	Operation
=	equal to
¬=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

These are used to connect legitimate arithmetic or character expressions (usually, only = and ¬= are used for the latter) to construct *conditional expressions* whose behavior resembles that of a one-bit string. This is, the final value of a conditional expression is either '1'B or '0'B. For example if X is an arithmetic variable of some kind, then the conditional expression

```
X**2<= 128
```

will have a value of '1'B if X^2 is less than or equal to 128 and '0'B if it is not. In fact, if A is a variable with the attribute BIT(1), we can actually write a statement like

```
A = X = 12;
```

which says, in essence, "place a '1'B in A if X equals 12; place a '0'B there if it doesn't." Note that the = sign is used in two different ways here.

Since conditional expressions behave this way, it is completely consistent to combine conditional expressions with logical operators to produce large conditional expressions. Thus, we can write an expression like

```
X = 12 & Y > 41
```

in which case it will produce a '1'B only when X is 12 and Y exceeds 41, i.e., both conditions must be true. Note that if we want to test for a condition where X and Y are both less than 57, for example, we *cannot* write

```
X & Y < 57
```

because it would be inconsistent with the properties outlined above. Instead, we must write

```
X < 57 & Y < 57
```

in order for PL/I to accept it.

Now that that is out of the way, we can look at the basic decision statement:

```
IF conditional expression THEN statement1;
statement2;
```

If *conditional expression* is true (produces '1'B) *statement*1 will be executed, followed by *statement*2 unless, of course, *statement*1 is a GO TO. If the condition is false, *statement*1 is ignored and the program goes directly to *statement*2.

EXAMPLE 95

Write a program to compute the sum of the integers 1–2,000, excluding those in the range 1,001–1,500.

Solution Program

```
EX95: PROCEDURE OPTIONS (MAIN);
  DECLARE TOTAL FIXED (7,0) INITIAL(1), INCR FIXED(4) INITIAL(2),
          TEST FIXED(4) INITIAL (1000);
ADDLOOP: TOTAL = TOTAL + INCR;
  IF INCR = TEST THEN GO TO ADJUST;
  INCR = INCR + 1;
  GO TO ADDLOOP;
ADJUST: IF TEST = 2000 THEN GO TO FINISH;
  INCR = 1501;
  TEST = 2000;
  GO TO ADDLOOP;
FINISH: END EX95;
```

18.7.2.1 The IF . . . ELSE Combination

The basic IF statement specifies an action if a condition is true, said action being bypassed if the condition is false. There are many occasions when it is desired to provide alternative actions for the true and false contingencies. To do this with our present repertoire we have to resort to separate GO TO statements. For example, suppose we want to add 12 to X if X is less than 20 and we want to subtract 10 if X is at least as large as 20. This can be done by the statements

```
        IF X < 20 THEN GO TO ADD12;
        X = X - 10;
        GO TO NEXT;
ADD12:  X = X + 12;
NEXT :  further processing;
```

PL/I provides a much more direct way via the ELSE statement. If we rewrite the above as

```
        IF X < 20 THEN X = X + 12;
        ELSE X = X - 10;
NEXT:   further processing;
```

the statement following THEN will be executed if X < 20, after which the program goes directly to the statement labeled NEXT. If X turns out to be >= 20, the statement following THEN is ignored and the statement following ELSE is executed, followed by the one labeled NEXT.

18.7.2.2 IF . . . ELSE Networks

The alternative statements following THEN and ELSE may themselves be IF statements, each with its own THEN and ELSE. This type of thing can continue as far as required. The easiest way to keep track of things is to treat IF . . . ELSE pairs as sets of parentheses and to exploit PL/I's flexible coding format.

EXAMPLE 96

A,B,C, and D are three-digit decimal integers. Set up the decision network for the following rules:

1. If A doesn't equal 4B, replace A with 1.1 times its previous value.
2. If A equals 4B, test C against 3A.
3. If C is less than 3A, test D against B. If D equals B, increase D by 4; if D exceeds 2B, decrease D by 1; for all other D's increase D by 1.
4. If C equals or exceeds 3A, test C against 4A.
5. If C exceeds 4A, decrease C by G; otherwise decrease C by 1.

Solution Procedure

```
EX96: PROCEDURE OPTIONS(MAIN);
      DECLARE (A,B,C,D) FIXED(3);

      .
      IF A = 4*B THEN
        IF C < 3*A THEN
          IF D=B THEN D = D+4;
          ELSE IF D>2*B THEN D = D-1;
               ELSE D = D+1;
        ELSE IF C > 4*A THEN C =C -6;
             ELSE  C = C-1;
      ELSE A = 1.1*A;
      END EX96;
```

18.7.2.3 The DO Statement

Although these IF . . . ELSE combinations allow the PL/I programmer to construct extremely complex decision networks, the specified activity for each contingency has been limited thus far to a single statement. When a particular action requires several statements, we could use GO TOS as before. PL/I offers a more organizationally consistent alternative in the form of the DO statement. This can be used to introduce a DO *group,* which

is a cluster of statements that are treated as an entity. When placed after a THEN or an ELSE, a DO group, whose conclusion is indicated by an END statement, is treated like a compound statement. For example, in the sequence

```
IF X < 2*Y THEN YES: DO:
                        X = X+3:
                        Y = Y-1:
                        END YES:
ELSE NO: DO:
         X = X-1:
         Y = Y+3:
         END NO:
NEXT: further processing:
```

the first DO group is executed if X is less than 2Y, after which the second is skipped. The reverse occurs when X is at least as large as 2Y. Note that the DO statement may be labeled. This often helps the programmer keep track of which END goes with which DO (or with which PROCEDURE).

18.7.3
The DO Loop

The DO statement can also be used to set up a mechanism which automatically goes through a loop a specified number of times. Processing conditions can be very carefully controlled by relatively simple specifications.

18.7.3.1
Simple Loops

The basic form for the DO loop is

```
label1:  DO index = v1 TO v2;

         .
         statements;

         .
         END label1;
```

index is the name of a variable (called the *index variable*) used to control the number of cycles through the loop. *v*1 is an integer (or an expression that works down to an integer) which sets the *starting value* for *index*. Similarly, *v*2 defines the *limiting value* for *index*. *index* does not have to be declared. PL/I automatically sets up storage for it and discards it after the loop cycling is finished.

EXAMPLE 97

Repeat Example 95 using DO loops. Recall that we are required to compute the sum of the integers 1–2,000, excluding those in the range 1,001–1,500.

Solution Program

```
EX97:  PROCEDURE OPTIONS(MAIN);
       DECLARE TOTAL FIXED (7,0) INITIAL(0);
LOOP1: DO J = 1 TO 1000;
       TOTAL = TOTAL + J;
       END LOOP1;
LOOP2: DO J = 1501 TO 2000;
       TOTAL = TOTAL + J;
       END LOOP2;
       END EX97;
```

Note that J was used as the index variable in both loops.

18.7.3.2 Multiple Specifications

Several *v*1 TO *v*2 phrases may be given in a single DO loop, allowing broken sequences.

EXAMPLE 98

Repeat Example 97 using a single DO loop.

Solution Program

```
EX98:  PROCEDURE OPTIONS(MAIN);
       DECLARE TOTAL FIXED (7,0) INITIAL (0);
LOOP:  DO J = 1 TO 1000, 1501 TO 2000;
       TOTAL = TOTAL + J;
       END LOOP;
       END EX98;
```

18.7.3.3 The BY Phrase

In the previous two examples we have allowed PL/I's default mechanism to take care of the number of cycles. That is, each time it went through the loop, J was increased by 1, until it exceeded the limiting value. We can specify whatever increment we want by adding a BY phrase:

DO *index* = *v*1 TO *v*2 BY *v*3;

*v*3 may even be negative, in which case *v*1 must exceed *v*2.

EXAMPLE 99

Compute the sum of all odd integers from 1 through 1,000 and all even integers from 1,501 through 2,000.

Solution Program

```
EX99: PROCEDURE OPTIONS(MAIN);
      DECLARE TOTAL FIXED 97,0) INITIAL(0);
      DO J = 1 TO 1000 BY 2, 1502 TO 2000 BY 2;
      TOTAL = TOTAL + J;
      END;
      END EX99;
```

Note that when J gets to 999, the loop is cycled and 2 is added to J, bringing it above the limit. At that point J is set to 1502 before the loop is cycled again. *v*3 need not be an integer.

18.7.3.4
The WHILE Phrase

Sometimes the cycling operation must be curtailed by the occurrence of a given condition. When the point at which this occurs cannot be anticipated, it must be tested for during each cycle. A convenient way to do this is with the WHILE clause:

DO *index* = *v*1 TO *v*2 BY *v*3 WHILE (*condition expression*);

As long as *condition expression* is true, the cycling will continue. If *v*2 is exceeded before the condition becomes false, of course the loop will stop. We shall show two alternative solutions in the next example to illustrate the contrast.

EXAMPLE 100

We have a one-dimensional array X consisting of one hundred 3-digit integers. Using the elements in sequence, find out how many elements can be added together before the sum exceeds 40,000. Use TOTAL to store the value of that sum.

First Solution Program

```
EX100A: PROCEDURE OPTIONS(MAIN);
        DECLARE (X(100), LAST) FIXED(3), TOTAL FIXED(5) INITIAL(0);
        .


        .
        DO I = 1 TO 100;
        TOTAL = TOTAL + X(I);
```

```
        IF TOTAL > 40000 THEN READY: DO;
                                        LAST = I - 1;
                                        TOTAL = TOTAL - X(I);
                                        GO TO OUTOF;
                                        END READY;
        END;
OUTOF: END EX100A;
```

Second Solution Program

```
EX100B: PROCEDURE OPTIONS(MAIN);
        DECLARE (X(100), LAST) FIXED(3), TOTAL FIXED(5) INITIAL(0);

        .
        DO I = 1 TO 100 WHILE (TOTAL <= 40000);
        TOTAL = TOTAL + X (I);
        LAST = I;
        END;
        END EX100B;
```

18.7.3.5 Nested DO Loops

DO loops may be placed inside one another as long as they don't overlap. When this *nesting* is used, the inner DO loop is cycled the specified number of times for each single cycle of the one containing it (just like the minute hand goes all the way around before the hour hand advances one unit).

EXAMPLE 101

X is a 50 × 200 array of 3-digit integers to be processed as follows:

1. All elements having odd row numbers and even column numbers are to be replaced by half their respective values if they are even or by 200 less than their current value if they are odd.
2. All other elements with values exceeding 500 are to be decreased by 100, the rest being left unchanged.

Solution Procedure

```
EX101:  PROCEDURE OPTIONS(MAIN);
        DECLARE X (50,200) FIXED(3);
```

```
OUTER: DO I = 1 TO 50;
INNER: DO J = 1 TO 200;
       IF MOD(I,2) = 0 & MOD(J,2) = 0
         THEN RULE1: DO;
                     IF MOD(X(I,J),2) = 0 THEN X(I,J) = X(I,J)/2;
                     ELSE X(I,J) = X(I,J) - 200;
                     END RULE1;
       ELSE IF X(I,J) > 500 THEN X(I,J) = X(I,J) - 100;
            ELSE;
       END INNER;
       END OUTER;
       END EX101;
```

Note that we could have omitted the DO and END from the DO group labeled RULE1 and simply set that up as a THEN IF. Also, we used an ELSE with nothing at all. This is called the *empty* or *null* ELSE. PL/I allows its inclusion to help the programmer keep his books balanced.

PROBLEMS

1. Write a program to compute the value of Q, given the following information:

$$Q = 2FG - 3.1 \log_e Z$$
$$G = 2Y^3 - 5.2Y - 7.1$$
$$F = 1/(X-2) + 1/(X-3)$$
$$X = 4$$
$$Y = 3$$
$$Z = 1.2$$

2. Write a program to accumulate the sum of the squares of the integers from 1 through 25.
3. Write a program to compute the median of an array of 200 floating-point numbers stored in memory.
4. Write a program to calculate the number of different ways a dollar bill can be broken into change. (1 half-dollar, 1 quarter, and 25 cents is one way; 2 quarters, 2 dimes, and 6 nickels is another.)
5. Given K as a positive integer argument, write a procedure to compute the value of K!, where $K! = K(K-1)(K-2) \ldots (1)$.
6. Given an $M \times N$ matrix A and an $N \times K$ matrix B in memory, write a program to compute their product. If C is the product matrix, it will have M rows and K columns and an element C_{ij} is obtained as

$$C_{ij} = \sum_{p=1}^{M} A_{ip} B_{pj}$$

7. Write a program that will take an integer X and produce a value Y according to the following relationship:

X	Y
0	0
1	0
2	0
3	1
4	1
5	1
6	2
7	2
8	2
9	3
10	3
11	3

8. Given an array W of 100 five-letter words, write a program to replace all vowels with asterisks.
9. Given an array V of 48 five-letter words organized into 8 rows and 6 columns, write a program that produces another array V1 consisting of four rows and six columns of 10-letter words. A new array member in row i and column j is obtained by combining alternate letters from the V members in the same column and in rows i and $i + 4$. For example, if $V_{2,5}$ is JXAOE and $V_{6,5}$ is UTPSD, then $V1_{2,5}$ will contain JUXTAPOSED.
10. Write a program that operates on a 30 × 30 array of floating point numbers and produces a new 30 × 30 array in which each element is obtained by squaring the element in the original array with a row 1 higher and a column 2 lower, such as $NEW_{i,j} = (OLD_{i+1,j-2})^2$. Make the subscript *circular* so that if a subscript goes above 30, the counting starts over (31→1); similarly, subscripts below 1 are renumbered (0→30).
11. B is a one-dimensional array of 100 bit strings, each with a length of 12 bits. Write a program that determines how many strings have twelve 1's, how many have eleven 1's, and so forth.
12. S1 and S2 are arrays like the one in Problem 11. Write a program that counts the number of matches in corresponding elements. If the score is more than 6 for a given pair, change all the 1's in S1's element to 0's and vice versa. Calculate the average number of matching bits in a pair of elements.

REFERENCES

1. Pollack, S. V., and T. D., Sterling, *A Guide to PL/I.* New York: Holt, Rinehart and Winston, Inc., 1969.

Chapter 19

PL/I Input/Output Programming

PL/I provides two basic types of input/output operations. These are designated as *stream*-oriented and *record*-oriented input/output. Both are supported by a repertoire of statements for operator/machine communication.

The record input/output statements are both file-oriented. Such files must be identified, defined, opened before use, and closed after use. OPEN and CLOSE statements are provided for these latter functions. In the stream input/output mode, the data are considered to be a continuous stream of characters; GET and PUT statements are used for input and output, respectively. In the record input/output mode, the data are considered to consist of discrete records. READ and WRITE statements cause a single record to be transmitted directly, without necessarily involving field conversions.

Mechanisms are available for defining and describing the files to be used in a program independent of the peripheral devices associated with these files. Each file is named by declaring its name with the FILE attribute; its usage is defined as INPUT, OUTPUT, UPDATE, or PRINT (a special type of OUTPUT); its organization is specified as STREAM or RECORD and additional information (such as access method and labeling) may be included. These various file-related attributes are supported by default mechanisms, as we shall see later. These extend to the basic input/output statements themselves. For example, an input statement without a file name will result in the use

of SYSIN. Similarly SYSPRINT is used in the absence of an output file designation.

19.1 STREAM-ORIENTED OPERATIONS

The stream input/output mode consists of three subdivisions: *list-directed* transmission, *data-directed* transmission, and *edit-directed* transmission. List-directed transmission permits the programmer to specify the variables to be transferred without specifying their format; data-directed transmission permits him to relate variable names and values on the input or output medium; and edit-directed transmission permits him to specify format, if he desires, by means of the FORMAT statement or format lists.

19.1.1 List-Directed Input/Output

List-directed input is obtained with a statement of the form

GET FILE (*filename*) LIST (*datalist*);

The list-directed output statement has the same form, with PUT replacing GET. File names and their attributes are defined in DECLARE and/or OPEN statements. Default files and attributes may be used, in which case the above form may be reduced to

GET LIST (*datalist*);

The elements of a data list are, in general, field names; in an output statement, however, string constants and expressions may also be included. Data-list elements are enclosed in parentheses and separated by commas.

When list-directed input is used, data fields are constants separated by commas or blanks. Four types of constants are recognized: real, character-string, bit-string, and complex. On output, fields are separated by blanks. In addition, the format of individual output fields depends on the internal precision and value of the particular items. The arrangement of output items can be controlled only to a relatively limited extent, most of the formatting being handled by standard mechanisms within PL/I. Usually, PL/I prints five items to a line unless otherwise specified, and we shall use that convention. We shall also assume that the standard input and output files are on the card reader and printer, respectively.

EXAMPLE 102

In each card of a data deck are four fields separated by commas. The first is a five-character alphameric field, and the second and third are numeric fields, given to the nearest thousandth and less than 1,000 in magnitude, and the last is a two-digit decimal integer. A typical card might be punched as follows:

'A1/C5', 95.109, −120.034, 26

Write a program to read each card and produce a printed line from each, consisting of the input fields, followed by the arithmetic mean of the numeric fields. After the last list has been printed, print a final line consisting of four fields. The first is the word TOTAL; the second and third are the sums of the first and second numeric fields of the data cards, respectively, and the fourth is the arithmetic mean of these sums. The first field of the last card is '**/**'.

Use list-directed input and output.

Solution Program

```
EX102:   PROCEDURE OPTIONS (MAIN);
         DECLARE (V1,V2)FLOAT(7), V3 FIXED(2), V4 CHARACTER(5),
         (T1,T2)FLOAT(8) INITIAL(0);
BRING:   GET LIST (V4,V1,V2,V3);
         PUT LIST (V4,V1,V2,V3,(V1+V2+V3), 3);
         T1 = T1 + V1; T2 = T2 + V2;
         IF V4 = '**/**' THEN GO TO NOMORE;
         ELSE GO TO BRING;
NOMORE:  PUT LIST ('TOTAL', T1,T2, (T1+T2)/2);
         END EX102;
```

The output line for the sample set of data would appear as

```
'TOTAL'  0.9510900E+02  -0.1200340E+03  26  0.358333E+00
```

Limited format control may be imposed on list-directed output by using combinations of the PAGE, SKIP and LINE options.

19.1.1.1 The PAGE Option

The keyword PAGE causes PL/I to start a new page immediately before printing, viz.,

```
PUT PAGE LIST (datalist);
```

If the programmer merely wishes to start a new page without printing anything (as is often done at the beginning of a program) he can simply write

```
PUT PAGE;
```

19.1.1.2 The SKIP Option

SKIP causes a new line to be started. The general form is

```
PUT SKIP(n) LIST (datalist);
```

where *n* indicates the number of lines to be skipped before printing. If (*n*) is omitted, PL/I assumes a value of 1.

19.1.1.3 The LINE Option

If we write

```
PUT LINE(n) LIST (datalist);
```

the items named in *datalist* will be printed starting on line number *n*. This is possible since PL/I keeps a line counter going, so that it can determine how many lines should be skipped to get to the specified line.

19.1.2 Data-Directed Input/Output

When the keyword DATA is used instead of LIST in a GET or PUT statement, the variables in the data list are associated with their respective names. For data-directed input, this means that each value must be accompanied by its name, the form being similar to that of an assignment statement. Thus a list of variables named in a GET DATA statement must correspond to a similar list in the input stream. For example, if we say

```
GET DATA (A,B,C,D);
```

and we assume that these variables have been duly declared (although that isn't always necessary), then A, B, C, and D values of 7, 206, 'ABXY' and −40.62, respectively, would appear in the input stream as

```
A = 7, B = 206, C = 'ABXY', D = -40.62;
```

Similarly, the statement

```
PUT DATA (A,B,C,D);
```

would produce a printed line in this form. Since the input variables are named in the data stream, they need not appear in the same order in which they are specified in the GET DATA statement. As a matter of fact, we can simply say

```
GET DATA;
```

and PL/I will read a sequence of items from the data stream until it comes to a semicolon.

When names are used in a GET DATA statement, the list must include (but need not be limited to) all of the items in the data stream (up to the next semicolon). Thus if the next items in a data stream are

```
X = 17.4, Y = 1006.2, Z = 'A7B', W = 61;
```

the statement

```
GET DATA (W,Z,X,Y,E);
```

will work, but something like

```
GET DATA (X,Y,W);
```

will not.

When the PUT DATA statement is used, PL/I's default format operates as in list-directed output (five items per line). The SKIP, PAGE, and LINE options can be used here as described before.

EXAMPLE 103

Repeat Example 102 using data-directed input/output. A typical data card might appear as follows:

```
V4 = 'A1/C5', V3 = 26, V1 = 95.109, V2 = -120.034;
```

Solution Program

```
EX103:   PROCEDURE OPTIONS(MAIN);
         DECLARE (V1,V2)FLOAT(7), V3 FIXED(3), V4 CHARACTER(5),
         (T1,T2,MEAN1,MEAN2)FLOAT(8) INITIAL(0);
BRING:   GET DATA(V1,V2,V3,V4);
         MEAN1 = (V1+V2+V3)/3;
         PUT DATA (V4,V1,V2,V3,MEAN1);
         T1 = T1 + V1; T2 = T2 + V2;
         IF V4 = '**/**' THEN GO TO NOMORE;
         ELSE GO TO BRING;
NOMORE:  PUT LIST ('TOTAL');
         MEAN2 = (T1 + T2)/2;
         PUT DATA (T1,T2,MEAN2);
         END EX103;
```

This time the output line for sample values will appear as

```
V4 = 'A1/C5', V1 = 0.9510900E+02, V2 = -0.1200340E+03,
V3 = 26, MEAN1 = 0.35833333E+00;
```

19.1.3 Edit-Directed Input/Output

This type of data transmission allows the programmer to describe his input or output format in complete detail. Consequently, it is the most versatile type of I/O. There is no need to show actual decimal points in numeric values, nor do character strings have to be enclosed by quote marks. Furthermore, since the format description also defines the exact positions of data items in the input stream, the various data items can be punched (or printed) next to each other, without intervening blanks or commas.

Edit-directed GET or PUT statements have the general form

```
GET
    EDIT (datalist) (formatlist);
PUT
```

datalist has its previous meaning and *formatlist* consists of a list of format specifications describing the portion of the data stream to be read or written.

19.1.3.1 The GET EDIT Statement

We shall look at PL/I's basic format specifications as used for edit-directed input, still assuming punched cards as the vehicle for the data stream. Bear in mind that PL/I is not concerned with the fact that there is a physical *break* at every 80th column. There is no correspondence between the end of a card and a stopping point in the data stream. By using buffers PL/I provides complete flexibility, so that it may appear as if we stopped reading in the middle of a card, with subsequent input being resumed at that point.

We shall deal with seven format specifications:

1. X(*n*) is used to indicate that the next *n* columns in the data stream are to be skipped, regardless of their contents. *n* may be any arithmetic expression that works down to an integer.
2. F(*m,n*) is used to describe fixed-point numerical values with *m* and *n* being used the same way as in a DECLARE statement. Thus, if a number is punched as 43296 and we want it to be stored and treated as 4329.6, our specification would be F(5,1). *m* and *n* may be expressions.
3. Floating-point numbers are described with the specification E(*m,n*), where *m,n* are used as before.
4. Complex numbers are specified by sequences of two F or E specifications as follows: C(F(*m,n*), F(*m,n*)), or C(E(*m,n*), E(*m,n*)). The first specification refers to the real part and the second takes care of the imaginary component.
5. The A specification, whose form is A(*n*), describes character strings. As stated before, no surrounding quote marks are required and there is no need to double up on quote marks internally. Thus the string ISN'T is punched exactly that way and is described simply as A(5).
6. Strings of 1's and 0's to be stored as bit strings are specified as B(*n*).
7. An alternate method, that of picture descriptions, may be used for format specification. These codes (Sec. 18.2.2.4) are used with the letter P and are bracketed by quote marks rather than parentheses. Thus a value punched as 43296 and described as F(5,1) could also be described as P'9999V9'.

EXAMPLE 104

Repeat Example 103, using data-directed output as before. This time, however, the input data are punched as follows:

Card Columns	*Item*	*Format*
6-10	V1	$XX_{(}XXX$
14-20	V2	$\pm XXX_{(}XXX$
21-22	V3	$XX_{(}$
76-80	V4	5 characters

A typical card would appear as in Figure 19.1.

Solution Program

```
EX104:   PROCEDURE OPTIONS (MAIN);
         DECLARE (V1,V2)FLOAT(7), V3 FIXED(2), V4 CHARACTER(5),
         (T1,T2,MEAN1,MEAN2)FLOAT(8) INITIAL(0);
BRING:   GET EDIT (V1,V2,V3,V4) (X(5), F(5,3), X(3) F(7,3), F(2), X(53), A(5));
         MEAN1 = (V1+V2+V3) 3;
         PUT DATA (V4,V1,V2,V3,MEAN1);
         T1 = T1 + V1; T2 = T2 + V2;
         IF V4 = '**/**' THEN GO TO NOMORE;
         ELSE GO TO BRING;
NOMORE:  PUT LIST ('TOTAL');
         MEAN2 = (T1+T2) 2;
         PUT DATA (T1,T2,MEAN2);
         END EX104;
```

Note that the format determines how the data are read in, but the form in which an item is ultimately stored is defined by the DECLARE, with PL/I taking care of any necessary conversions.

In this example we selected our data-card layout in such a way that the end of our data list coincided with the end of a card. Thus, when the program branched back to BRING to read the next set of data, the input stream was automatically "positioned" so that the next column was column 1 of the next data card. Hence the x(5) specification would once again bring the stream to column 6 of that next card, which is where we want it to be.

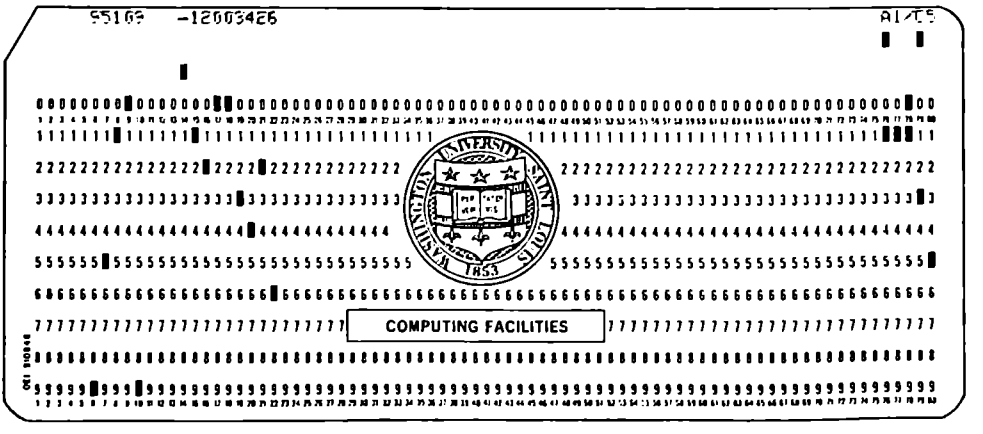


Figure 19.1

Suppose however, that the situation were altered so that V4 is punched in columns 71-75. Then if our input statement read

```
BRING: GET EDIT (V1,V2,V3,V4) (X(5), F(5,3), X(3), F(7,3), F(2), X(48),
A(5));
```

execution of this statement for the first data card would leave the input stream positioned ready to deliver column 76 of that card. Thus, when the GET EDIT statement came around again, we would skip the first five columns as before, but these would be columns 76-80 of the first card rather than columns 1-5 of the next one. Unfortunately, the obvious correction

```
BRING: GET EDIT (V1,V2,V3,V4) (X(5), F(3,3), X(3), F(7,5), F(2), X(48),
A(5), X(5));
```

will not work. PL/I ignores the last format specification if it is an X-specification. There are several ways to get around this, some more elegant than others. The most convenient of these is to use the SKIP option by placing it at the desired spot in the format list. This immediately positions the input stream at the first column of the next card. For our example,

```
BRING: GET EDIT (V1,V2,V3,V4) (X(5), F(5,3), X(3), F(7,5), F(2), X(48),
A(5), SKIP);
```

19.1.3.2 The PUT EDIT Statement

The various format descriptions work here as described for edit-directed input, serving in this case to define the output layout in complete detail. These may be combined with the following specifications to produce a virtually limitless variety of output descriptions including multiline and multipage formats:

1. The PAGE, SKIP and LINE options may be used in combination with other specifications. For example, suppose we wanted to produce the following output:
   (a) The current value of A in the form XXX.XX starting in the 11th column of the first line of a new page.
   (b) The current value of B in the form 0.XXXXE+XX starting in column 30 of that same line.
   (c) The current value of C, a seven-character string starting in column 7 of the third line.
   (d) The current values of Y and Z, both of which are unsigned four-digit integers, in columns 1-4 and 51-54 on the second line of the next page.

   What with SKIP and LINE, PICTURE versus A, F, or E specifications, etc., we could specify the output any number of ways. Let us show just two:

```
PUT PAGE EDIT(A,B,C,Y,Z)X(10),F(7,2),X(12),E(11,4),SKIP(2),
X(6),A(7),PAGE,SKIP,F(4),X(46),F(4));
PUT EDIT(A,B,C,Y,Z)(PAGE,X(10),PICTURE'$999.99',X(12),E(11,4),
LINE(3),X(6),A(7),PAGE,LINE(2),PICTURE('9999'),X(46),F(4));
```

Note that the numerical specifications must include room for the decimal point, sign, etc. Thus, B is specified as E(11,4) to accommodate the exponent (2 digits), its sign (one digit), the E (1 digit), the fraction (4 digits), the decimal point (1 digit), the zero (1 digit) and the value sign.

2. The programmer may use the COLUMN option to save some counting. Replacing the X specifications will change the second version in our example to

```
PUT EDIT (A,B,C,Y,Z)(PAGE,COLUMN(11),PICTURE'$999.99',
COLUMN(30),E(11,4),LINE(3),COLUMN(7),A(7),PAGE,LINE(2),
PICTURE'9999',COLUMN(51),F(4));
```

3. Literal values may be included in an edit-directed output specification by specifying them as part of the data list and accounting for them with appropriate A-specifications. To illustrate, let us add some requirements to the previous example: The seven-character string, item (c), is to be followed by two blanks and the message IS THE NAME OF THE LATEST CRAFT.. Also, preceding the two items in (d), on the first line of that new page, add the heading SECOND OUTPUT PAGE centered on the line.

The first message is 34 columns long, including the two blanks before IS. The heading requires 18 columns, so that we would leave (132–18)/2 or 57 blank columns on each side. Our statement will read

```
PUT EDIT(A,B,C,'␣␣IS THE NAME OF THE LATEST CRAFT.',
'SECOND OUTPUT PAGE',Y,Z)
PAGE,X(10),F(7,2),COLUMN(30),E(11,4),LINE(3),X(6),A(7),A(34),PAGE,
X(57),A(18),LINE(2),F(4),COLUMN(51),F(4));
```

If we are not interested in keeping track of exactly where each item is to be printed, we can let PL/I do the counting. In our example the A(34) and A(18) can be replaced by A(1) and A(1). That is, it is sufficient merely to show that a character field is present. PL/I will determine the exact count by examining the data list.

19.1.3.3 Additional Editing Capabilities with the PICTURE Specification

As in COBOL, the PICTURE specification triggers all sorts of editing mechanisms which include automatic searches for special items (such as high-order zeros), insertion of special characters, and so on. These are

usually employed for financial types of output in which dollar figures or amounts on checks must be expressed in certain forms.

We have already used the 9, K, V, A, and X codes (Sec. 18.2.2.4). In addition, we have implied (in the format example of the previous section) that the use of a . specifies the actual appearance of a period. We shall repeat these for continuity, followed by additional codes:

1. Numeric Codes
   9 specifies a number in that position.
   V shows the position of an assumed decimal point.
   . shows the position of an actual (physical) decimal point.*
   , shows the position of an actual comma.*
   / shows the position of an actual slash.*
   B specifies a blank in that position.
   K specifies the beginning of the exponent in a floating-point number.
   F specifies that the following decimal integer, optionally signed and enclosed in parentheses, is the scaling factor. The scaling factor specifies that the assumed position of the decimal or binary point should be that many places to the right (or to the left, if negative) of the position assumed in the absence of the scaling factor.
2. String Characters
   A specifies the position of an alphabetic character or blank.
   X specifies the position of any character.
   9 specifies the position of a decimal digit or blank.
3. Sign Characters
   Numeric data may be submitted with the sign overpunched in a particular column. Similarly, if the output is to be punched, the programmer may define the position of the overpunch.

   T shows the position to be overpunched with the sign.
   I shows the position to be overpunched only if the sign is +; otherwise there is no overpunch.
   CR (for *credit*) indicates the positions in which the letters CR are to appear when a result is less than zero.
   DB (for *debit*) is used the same way as CR.
4. Zero Suppression Characters
   Z replaces that position with a blank if it is a leading zero; otherwise nothing happens.
   * works like Z, but uses an asterisk instead of a blank.
   Y works like B, replacing zeros with blanks.
5. Drifting Editing Characters
   $ specifies position of $ if used singly. If used in a drifting string, the $ appears in the rightmost leading zero position, and the other leading

*These will appear only if there is a significant digit ahead of the indicated position. Otherwise, a blank will be inserted there, or a drifting character will be shown, if specified.

zeros are suppressed. For example P'$$$$.99' for a value of 7.64 will produce $7.64.
s specifies position of + if the field is $\geqslant 0$ and the position of − if the field is < 0. If used in drifting string, usage is similar to that of $ string. + specifies position of + if the field $\geqslant 0$, blank, otherwise. If used in drifting string, usage is similar to that of S string.
− specifies position of − if the field < 0, blank, otherwise. If used in drifting string, usage is similar to that of S string.

19.1.3.4 Additional Format Features

We have already indicated that any format specification may be a variable or expression. If that value works out to be less than 1, that specification is ignored.

Repeating specifications may be written in shortened form by means of factoring. For example, the statement

```
PUT EDIT (X,Y,Z,W,A,B,C,D)(X(7),F(2),F(5,2),X(8),A(3),X(3),
F(2),F(5,2),X(8),A(3),Z(3),F(5),X(3),F(5));
```

can be shortened to

```
PUT EDIT (X,Y,Z,W,A,B,C,D)(X(4),2(X(3),F(2),F(5,2),X(8),
A(3)),2(X(3),F(5)));
```

Similarly, the statement

```
GET EDIT (A,B,C,D,E,F,G,H,X,Y)(X(2),F(3),X(2),F(3),X(2),F(3),
X(2),F(3),X(2),F(3),X(2),F(3),X(2),F(3),X(2),F(3),X(2),F(3),X(2),F(3));
```

can be shortened, in this case, to

```
GET EDIT (A,B,C,D,E,F,G,H,X,Y)(X(2),F(3));
```

When several statements in a program require identical format specification lists, *remote* formats allow one format to be referenced from various places in the program. This is done by writing the GET or PUT statement as

```
PUT
GET EDIT (datalist) (R(label));
```

with *label* referring to a statement of the form

```
label: FORMAT (formatlist);
```

Of course, the data lists in the various statements must be consistent with the descriptions in the FORMAT statement, but need not be identical to each other.

19.1.4
The GET STRING Statements

PL/I allows the programmer to set up an area in storage which can then be used as an "input" source. In this way, he can look at the input after it has been brought in, assigning format specifications after the data have been inspected. Although any stream-oriented form may be used, edit-directed is the most versatile. The form is

GET STRING (*stringname*) EDIT (*datalist*) (*formatlist*);

EXAMPLE 105

Each of a series of data cards contains values for X, Y, and Z in one of three possible formats. A single digit, punched in column 80 of each card indicates the type of format used:

Format 1			*Format 2*			*Format 3*		
Column(s)	*Item*	*Format*	*Column(s)*	*Item*	*Format*	*Column(s)*	*Item*	*Format*
5-6	X	XX	1-4	X	XXXX	1-3	X	XXX
10-12	Y	$X_{(3)}$XX	5-6	Y	XX	51-54	Y	$XXX_{(3)}$X
32-36	Z	$X_{(3)}$XXXX	7-10	Z	$_{(3)}$XXXX	58	Z	$_{(3)}$X

Write a series of statements to read in the data.

Solution Procedure

We shall set up an array of labels and use the indicator in column 80 as a subscript to specify which FORMAT statement to use.

Solution Program

```
EX105:  PROCEDURE OPTIONS(MAIN);
        DECLARE LOOKHERE CHARACTER(79), HERE(3)LABEL, SIGNAL
        FIXED(1), X FIXED(4), Y FIXED(5,2), Z FIXED(5,4);
        GET EDIT (LOOKHERE,SIGNAL)(A(79),F(1));
        GET STRING (LOOKHERE) EDIT (X,Y,Z) (R(HERE(SIGNAL)));
HERE(1): FORMAT(X(4),F(2),X(3),F(3,2),X(19),F(5,4));
HERE(2): FORMAT (F(4),F(2),F(4,4));
HERE(3). FORMAT(F(3),X(47),F(4,1),X(3),F(1,1));
        more statements;
```

19.1.5
Stream-Oriented Arrays and Structures

Entire collections of data may be read or written with appropriate GET or PUT statements using lists, data-, or edit-directed forms. This is a considerable convenience but it must be used carefully, that is, with full knowledge of PL/I's default sequence for storing individual elements.

19.1.5.1
Input/Output of Arrays

If X is defined by the statement

```
DECLARE X(4,5) FIXED(3);
```

the statement

```
GET LIST(X);
```

will bring in the next twenty values from the input stream, storing them respectively in X(1,1), X(1,2), X(1,3), and so on. The corresponding PUT statement will print all twenty in the same order. GET/PUT EDIT would operate similarly. In the case of a data-directed array, of course, the order would be determined by the element's position in the input stream, since each element would be labeled separately. Furthermore, the statement

```
GET DATA(X);
```

would not require the presence of all twenty elements in the input stream.

Parts of arrays and different sequencing of array elements can be handled in list- and edit-directed forms by including a DO statement as part of the GET or PUT instruction. Using Y as a ten-element one-dimensional array, we can illustrate the general form with the statement

```
GET LIST ((Y(I) DO I = 1 TO 10));
```

The extra set of parentheses is required. If we were writing only the odd-numbered elements of Y, we could say

```
PUT LIST ((Y(I) DO I = 1 TO 10 BY 2));
```

Using X again as a 4×5 array, the statement

```
GET EDIT(((X(I,J) DO J = 1 TO 5) DO I = 1 TO 4))(X(1),F(3));
```

is equivalent to

```
GET EDIT (X)(X(1),F(3));
```

On the other hand, if we wrote

```
GET EDIT(((X(I,J) DO I = 1 TO 4) DO J = 1 TO 5))(X(1),F(3));
```

the next twenty values would be read in and stored as X(1,1), X(2,1), X(3,1), X(4,1), X(1,2), X(2,2), etc. Thus the DO statement closest to the data list corresponds to the *innermost* loop in a nested series of loops. Note that no END statement is required to close such a loop in an input/output statement.

19.1.5.2 Input/Output of Structures

Entire structures may be transmitted with single GET statements, as long as the order of elements in the input stream follows that specified in the declaration. The same holds true for minor structures. Individual elements may be handled as well by using their qualified names in the data list. To avoid confusion, it is generally a good idea to use fully qualified names for such purposes.

19.2 FILE DESCRIPTION

As indicated before, the forms we have been using for the GET and PUT statements represent special versions of the more general

```
GET                    LIST
PUT FILE (filename)    DATA    ....
                       EDIT
```

Implied in our usage has been a set of default operations which allow PL/I to assign standard files with prescribed attributes. Now that we have examined the basic techniques for handling stream-oriented data transmission, we must acquaint ourselves with the general properties of such files. In addition, the use of explicitly described files is all-important for record-oriented input/output.

19.2.1 Declaration of Files

Files are named, like variables, and described in DECLARE statements by a list of specific attributes. Although the FILE attribute is basic, its assignment is usually handled by default, based on the presence of other attributes.

There are three usage attributes: INPUT (default), OUTPUT or UPDATE, the latter allowing a file to be used for both input and output. The data form is specified either as STREAM or RECORD. Although STREAM is the default option, the specification of certain other attributes will often determine whether STREAM or RECORD is appropriate. We shall look at these implied file attributes in a little while. Just as GET and PUT are statements designed exclusively for stream-oriented transmission, READ and WRITE are used exclusively in conjunction with RECORD files.

Access to record-oriented files is either SEQUENTIAL or DIRECT. SEQUENTIAL specifies that the order in which records are accessed is determined by the physical organization of the data file; DIRECT specifies that such order depends on the value of a designated *key*. Files with the DIRECT attribute must also have the KEYED attribute.

SEQUENTIAL RECORD files may be either BUFFERED or UNBUFFERED. Files with the STREAM attribute and those with the DIRECT attribute may be neither BUFFERED nor UNBUFFERED. A BUFFERED file is one which passes through an intermediate memory region (or *buffer*) on the way between an input/output device and the memory region designated by the declared variables. This intermediate region is made accessible to the programmer by means of variations in the input/output statements.

A SEQUENTIAL file may or may not be KEYED—that is, be declared to be KEYED(*n*), where *n* is a decimal integer constant giving the length of the key in characters. If it is KEYED, the keys may be ignored while accessing sequentially, or they may be extracted from the file or placed into the file by the KEYTO *and* KEYFROM options. It is possible to create a KEYED SEQUENTIAL OUTPUT file and later to access it as a DIRECT file.

SEQUENTIAL INPUT and SEQUENTIAL UPDATE files may be positioned to a particular record by a READ statement that specifies the key of the desired record. Thereafter, successive READ statements without the KEY option will access the records sequentially. This kind of access may be used only if the file has keyed records and the KEYED attribute.

Records of a SEQUENTIAL UPDATE file may be rewritten, modified, or ignored, but the number or records cannot be changed. With a DIRECT UPDATE file, however, records may be added to the file through use of the WRITE statement or deleted from it through use of the DELETE statement. An existing record in an UPDATE file may be replaced through use of a REWRITE statement.

A special type of OUTPUT file is one whose contents are eventually to be printed, irrespective of the device on which that file is initially stored. Such a file is given the PRINT attribute, which obviates the need for specifying OUTPUT. Also, the PRINT attribute is restricted to STREAM files.

Table 19.1 summarizes the declared file attributes, and Table 19.2 shows the relationships between declared and implied attributes.

TABLE 19.1 DECLARED FILE ATTRIBUTES

Choice	Default
INPUT, OUTPUT, UPDATE	INPUT
STREAM, RECORD	STREAM
SEQUENTIAL, DIRECT	SEQUENTIAL
BUFFERED, UNBUFFERED	BUFFERED
PRINT*	
KEYED*	
BACKWARDS*	

*These must be explicitly declared.

Thus the statement

```
DECLARE X FILE INPUT RECORD SEQUENTIAL BACKWARDS,
Y FILE PRINT OUTPUT STREAM;
```

TABLE 19.2 IMPLIED FILE ATTRIBUTES

If Declaration Specifies	PL/I will also Assign
UPDATE	RECORD
SEQUENTIAL	RECORD
DIRECT	RECORD
BUFFERED	RECORD, SEQUENTIAL
UNBUFFERED	RECORD, SEQUENTIAL
PRINT	OUTPUT, STREAM
BACKWARDS	RECORD, SEQUENTIAL, INPUT
EXCLUSIVE	RECORD, DIRECT, UPDATE
KEYED	RECORD

can be shortened to

```
DECLARE X BACKWARDS, Y PRINT;
```

EXAMPLE 106

Repeat Example 104 using PICKUP as an edit-directed input file and OUTGO as an edit-directed print file. Each output line is to appear as follows:

Starting Column	*Item*	*Form*
11	V4	five characters
30	F1	XX.XXX
50	V2	XXX.XXX
70	V3	XX
90	MEAN1	XXX.XXXXX

The last line, consisting of the word TOTAL, followed by T1, T2, and MEAN2, is to be written in data-directed mode.

Solution Program

```
EX106:  PROCEDURE OPTIONS(MAIN);
        DECLARE (V1,V2) FLOAT(7),V3 FIXED(2),V4 CHARACTER(5),
        (T1,T2,MEAN1,MEAN2)FLOAT(8) INITIAL(0), PICKUP FILE INPUT
        STREAM, OUTGO FILE PRINT;
BRING:  GET FILE(PICKUP) EDIT(V1,V2,V3,V4)(X(5),F(5,3),X(3),F(7,3),F(2),
        X(53),A(5));
        MEAN1 =(V1+V2+V3)/3;
        PUT FILE (OUTGO) EDIT (V4,V1,V2,V3,MEAN1) (SKIP,COLUMN(11),
        A(5),COLUMN(30),F(7,3)COLUMN(50),F(8,3),COLUMN(70),F(2,0),
        COLUMN(90),F(11,5));
        T1 = T1 + V1; T2 = T2 + V2;
        IF V4 = '**/**' THEN GO TO NOMORE;
        ELSE GO TO BRING;
NOMORE: PUT SKIP FILE (OUTGO)('TOTAL');
        MEAN2 = (T1+T2)/2;
        PUT FILE(OUTGO) DATA (T1,T2,MEAN2);
        END EX106;
```

19.2.2
Activation and Deactivation of Files

Declaration of a file provides PL/I with definitive information about its attributes but does not make that file available for use. This is done explicitly by means of the OPEN statement, whose basic form is

OPEN FILE (*filename*) *attr*1, *attr*2, ...

where *attr*1, *attr*2, ..., refer to a list of file attributes including but not limited to those specifiable in the DECLARE statement. That is, it is possible to handle both the declaration and activation in a single OPEN statement, for example, the statements

```
DECLARE KEEPIT INPUT SEQUENTIAL UNBUFFERED;
OPEN FILE (KEEPIT);
```

can be replaced by

```
OPEN FILE (KEEPIT) INPUT SEQUENTIAL UNBUFFERED;
```

When the file being opened has the PRINT attribute, special additional definitions may be given. These are the PAGESIZE and LINESIZE options. These appear in the form

OPEN FILE (*filename*) *attr*1, *attr*2, . . . PAGESIZE (*expression*) LINESIZE (*expression*);

The numeric expression following PAGESIZE indicates the number of lines to appear on each output page; similarly, the LINESIZE option defines the number of available print spaces on an output line. Default values are used if either or both are not specified for a particular file (55 lines per page and 120 spaces per line are typical default values).

Files need not always be opened explicitly. If an OPEN statement is not included for a file and all its desired attributes are defined or inplied, it will automatically be opened when the program reaches the first input/output statement in which it is involved.

The CLOSE statement deactivates a file. If the file happens to be on tape, it will be repositioned to its first record. All files are automatically closed at the end of the program.

19.2.3
Labeling of Files—The IDENT Option

File labels are handled by means of the IDENT option which is specified as part of the OPEN or CLOSE statement. IDENT takes one argument (enclosed in parentheses) whose usage is determined by the file's attributes. In general, the IDENT option, when used with an input file, specifies the name of a character string variable into which the file's label will be placed. When

TABLE 19.3 THE IDENT OPTION

Type of Statement	File Attributes	IDENT Activity
OPEN	INPUT	Reads header label
	INPUT, BACKWARDS	Reads trailer label
	UPDATE	Reads header label
	OUTPUT	Writes header label
CLOSE	INPUT	Reads trailer label
	INPUT, BACKWARDS	Reads header label
	OUTPUT	Writes trailer label
	UPDATE	Writes trailer label

used with an output file, it is treated as a character expression which will become that file's label. Table 19.3 summarizes the properties of the IDENT option.

19.3 RECORD-ORIENTED OPERATIONS

When the RECORD attribute is used, file records are read into and written out of memory areas defined as unsubscripted level 1 structures. These are organized to coincide with the components of the data record so that the READ or WRITE statements are not designed to carry detailed format information.

19.3.1 The ON Statement

One of the features of PL/I which brings new flexibility to the source language debugging process is the ON statement. Its general form is

ON *condition statement;*

where *statement* is some executable instruction (often a GO TO) and *condition* is one of a series of recognizable situations which may prevent continued execution or alter it in some unknown way unless some special action is taken. These conditions are analogous to some of the interrupts that System/360 is designed to handle (see Chapter 3), including various types of internal anomalies as well as those associated with input/output. When a particular condition is encountered, such as an attempt to divide by zero, the system automatically takes some type of action (which may or may not please the user). The ON statement allows him to branch to a set of instructions which specify his own action. We shall concern ourselves with file-related interrupt conditions, deferring the others until the next chapter.

ENDFILE (*filename*) is a condition raised by attempting to read (with GET or READ) past the end of the file named. It remains until the file is closed.

ENDPAGE (*filename*) is a condition raised by the PUT statement when an attempt is made to start a new line beyond the limit specified for the PRINT file by the PAGESIZE option in the OPEN statement. Standard system action is to start a new page.

TRANSMIT (*filename*) may be raised during any input/output operation and is caused by a permanent transmission error on the specified file.

UNDEFINEDFILE (*filename*) (abbreviated UNDF) is raised on any OPEN statement if the named file cannot be opened:

NAME (*filename*) is a condition that may be raised on data-directed GET statements. It is caused by an unrecognizable identifier in the input or by an identifier not in the associated data list. The standard system action is to ignore the field and to issue a comment.

KEY (*filename*) may be raised in any of the following cases:

1. A READ for which the key is not found.
2. A WRITE or LOCATE for which the key already exists.
3. A REWRITE for which the key is not found.
4. A DELETE for which the key is not found.

RECORD (*filename*) is a condition that may be raised by any READ or REWRITE operation. It is caused by a record containing more or less data than the specified variable.

For those of the above ON-conditions in which the standard system action has not already been specified, such action is to raise the ERROR condition and to issue a comment to the operator.

19.3.2 Handling Unbuffered Files

The basic input/output statements for record-oriented files with the UNBUFFERED attribute are

```
READ FILE (filename) INTO (v1);
WRITE FILE (filename) FROM (v1);
```

where *v*1 is declared so that it conforms to the record's format. An unbuffered file may be declared with the KEYED attribute:

```
DECLARE filename RECORD KEYED (n);
```

where *n* defines the number of characters in the key. When the file is also SEQUENTIAL and is declared for input or update purposes, we can say

```
READ FILE (filename) INTO (v1) KEYTO (v2);
```

in which case the key will be read into *v*2, from whence it can be used as any other character string. The IGNORE option is also available for such files, its usage being as follows:

```
READ FILE (filename) IGNORE (expression);
```

The expression is evaluated to an integer and that many records are skipped. The next available record can then be brought in with another READ statement.

When writing onto an unbuffered output or update file, the WRITE statement may take the form

```
WRITE FILE (filename) FROM (v1) KEYFROM (v2);
```

in which case v2's contents will become the key for that record.

EXAMPLE 107

A deck of 80-character card records is to be converted to a keyed sequential file on magnetic tape with each ten card records forming a single physical tape record. Use consecutive integers as keys. Write a program to perform this conversion, using the UNBUFFERED and RECORD attributes. A six-digit key will suffice.

Solution Program

```
EX107:    PROCEDURE OPTIONS (MAIN);
          DECLARE CARDS FILE RECORD INPUT SEQUENTIAL UNBUFFERED,
            TAPE FILE RECORD OUTPUT SEQUENTIAL UNBUFFERED KEYED(6);
          DECLARE 1 INHERE CHARACTER(80),
            1 OUTHERE,
              2 OUTCARD(10) CHARACTER(80),
            KEYVAR FIXED(6) INITIAL(0);
          ON ENDFILE (CARDS) GO TO FINISH;
          ON TRANSMIT (TAPE) GO TO BADTAPE;
          ON UNDEFINEDFILE (TAPE) GO TO BADTAPE;
          ON KEY (TAPE) GO TO BADTAPE;
          ON RECORD (CARDS) GO TO BADCARD;
          OPEN FILE (CARDS), FILE (TAPE);
LOOP:     DO I = 1 TO 10;
          READ FILE (CARDS) INTO (INHERE);
          OUTCARD(I) = INHERE;
          END LOOP;
          KEYVAR = KEYVAR + 1;
          WRITE FILE (TAPE) FROM (OUTHERE) KEYFROM (KEYVAR);
          GO TO LOOP;
BADTAPE:  PUT SKIP LIST ('BAD TAPE AT KEY', KEYVAR);
          GO TO FINISH;
BADCARD:  PUT SKIP LIST ('BAD CARD. LAST KEY WAS', KEYVAR);
FINISH:   CLOSE FILE (CARDS), FILE (TAPE);
          END EX107;
```

19.3.3 Handling Buffered Files

System/360's capabilities for overlapped operations may be exploited to some extent in PL/I by using buffered files, in which case record-oriented data transmission takes place to and from areas in memory

which differ from those in which processing occurs. Thus the next record (or two records, or *n* records), depending on specifications to the system) are internally available for processing while subsequent records are being brought in.

PL/I's basic mechanisms for setting up and maintaining buffers are activated by assigning the BUFFERED attribute to each file to be handled in this manner. Since the location of the buffer is out of the programmer's control, he provides only its description and defines a *pointer* variable which PL/I will set to the buffer's current address on demand. The buffer description is supplied in an ordinary DECLARE statement. Its name is given the CONTROLLED attribute and it is said to be a *based* variable, meaning that its location may shift from time to time. We still describe the record area as a structure, but this time we add these aforementioned attributes. For example, if we wish to describe an area called PLACE consisting of a five-element array of three-digit integers (called NUM) and a twelve-character field called NAME, said area's address to be stored in WHERE, we would say

```
DECLARE  1 PLACE CONTROLLED (WHERE),
         2 NUM(5) FIXED(3,0),
         2 NAME CHARACTER(12);
```

The input statement for buffered files has the general form

```
READ FILE (filename) SET (pointer);
```

This gets the data into the buffer and stores its address in *pointer*. Now that record, or any part of it, may be referred to by using the appropriate qualified name in the structure.

Output is handled with one of two types of statements. The WRITE statement (with or without the KEYFROM option) is used as shown in the previous section. As an alternative, it is possible to trigger a slightly different activity with the LOCATE statement.

The LOCATE statement does not immediately cause data to be written. Instead it causes allocation of memory for a buffer having the same attributes as the specified based variable. In addition, data are placed in this buffer but not written out until immediately before the next LOCATE or WRITE statement involving the same file or immediately prior to the closing of that file.

EXAMPLE 108

Repeat Example 107 using buffered files.

Solution Program

```
EX108:   PROCEDURE OPTIONS (MAIN);
         DECLARE CARDS FILE RECORD INPUT SEQUENTIAL BUFFERED,
           TAPE FILE RECORD OUTPUT SEQUENTIAL BUFFERED KEYED(6);
```

```
        DECLARE 1 INHERE CHARACTER(80) CONTROLLED (INPUT),
                1 OUTHERE CONTROLLED (OUTPUT),
                  2 OUTCARD(10) CHARACTER(80),
                KEYVAR FIXED(6) INITIAL(0);
        ON ENDFILE (CARDS) GO TO FINISH;
        ON TRANSMIT (TAPE) GO TO BADTAPE;
        ON UNDEFINEDFILE (TAPE) GO TO BADTAPE;
        ON KEY (TAPE) GO TO BADTAPE;
        ON RECORD (CARDS) GO TO BADCARD;
        OPEN FILE (CARDS), FILE (TAPE);
LOOP:   DO I = 1 TO 10;
        READ FILE (CARDS) SET (INPUT);
        OUTCARD(I) = INHERE;
        END LOOP;
        KEYVAR = KEYVAR + 1;
        LOCATE (TAPE) SET (OUTPUT) KEYFROM (KEYVAR);
        GO TO LOOP;
BADTAPE: PUT SKIP LIST ('BAD TAPE AT KEY', KEYVAR);
        GO TO FINISH;
BADCARD: PUT SKIP LIST ('BAD CARD. LAST KEY WAS', KEYVAR);
FINISH: CLOSE FILE(CARDS), FILE (TAPE);
        END EX108;
```

19.3.4 Handling Update Files

Files with the UPDATE attribute (that is, files which can be used for input and output in the same program) are declared, opened, read from, and closed in the same way as other files. To help PL/I handle output efficiently, the REWRITE statement is used for writing instead of the WRITE statement. Aside from substituting REWRITE for WRITE, all other phrases and options are unchanged.

EXAMPLE 109

A sequential file consists of 200-character records. Each record has the following format:

Positions	*Field Description*
1-2	Year code (numeric)
3-4	Month code (numeric)
5-10	Status code (numeric)
11-200	Other fields (alphameric)

Write a program to update the file in the following way:

1. Set the status code to 999999 if the record date is more than six months prior to the current date, and rewrite the record. The current month and year are read in from a card in list-directed mode.
2. Ignore all other records.

Solution Procedure

A simple technique is used to compare two dates given as month and year. If YC and YS are the current and stored years and MC and MS are the corresponding months, the number of elapsed months is 12*(YC − YS) + (MC − MS). (All is well unless a turn of the century is involved.)

Solution Program

```
EX109:  PROCEDURE OPTIONS (MAIN);
        DECLARE FIXUP FILE UPDATE SEQUENTIAL BUFFERED RECORD,
          1 INPLACE CONTROLLED (ARROW),
          2 YEAR FIXED(2),
          2 MONTH FIXED(2),
          2 STATUS FIXED(6),
          2 OTHER CHARACTER(190),
          (YC, MC) FIXED(2);
        ON ENDFILE (FIXUP) GO TO FINISH;
        GET LIST (MC, YC);
LOOP:   READ FILE (FIXUP) SET (ARROW); /* NOTE THAT FIXUP IS
        OPENED BY THE READ */
        IF 12*(YC − YEAR) + MC − MONTH <= 6 THEN GO TO LOOP;
        STATUS = 999999;
        REWRITE FILE (FIXUP) FROM (INPLACE);
        GO TO LOOP;
FINISH: CLOSE FILE (FIXUP);
        END EX109;
```

19.3.5 Direct Files

Direct files are handled pretty much the same way as sequential files, except that their definition (presented externally to the supervisory system) is more elaborate. Declared attributes have been described previously.

Direct files with the UPDATE attribute are more flexible than their sequential counterparts. Not only may records be replaced by means of the REWRITE statement, they may also be added by means of the WRITE statement and deleted via the DELETE statement. The basic form for this statement is

```
DELETE FILE (filename) KEY (keyname);
```

in which case the value of *keyname* will be used as a search criterion to identify the record to be deleted.

EXAMPLE 110

A direct file consists of 200-character records, each with the following format:

Positions	*Contents*	*Format*
1-5	Key	XXXXX
6-35	Name	Characters
36-65	Address	Characters
66-95	City, State, Zip Code	Characters
96-200	Other fields	Characters

Write a program to update the file according to a deck of correction cards with the following format:

Column(s)	*Contents*	*Format*
1-6	Key	XXXXX
6	Action code	X

where 0 = delete record with same key
1 = replace name with name on this card
2 = replace address with address on this card
3 = replace city, state, and zip code with those on this card
4-9 = no action

7-36	Correction fields	Characters
37-80	Other fields	Characters

Solution Program

```
EX110:  PROCEDURE OPTIONS (MAIN);
        DECLARE MASTER FILE UPDATE DIRECT RECORD KEYED(5),
          CARDS FILE INPUT SEQUENTIAL RECORD UNBUFFERED,
          1 AREA, 2 KEY FIXED(5), 2 FIELD(3) CHARACTER(30),
            2 OTHER CHARACTER(105),
          1 INHERE, 2 KEY FIXED(5), 2 ACTION FIXED(1),
            2 CORRECTION CHARACTER(30),
            2 OTHER CHARACTER(44);
        ON ENDFILE (CARDS) GO TO FINISH;
LOOP:   READ FILE (CARDS) INTO (INHERE);
        IF ACTION = 0 THEN DO;
          DELETE FILE(MASTER)KEY(INHERE.KEY);
          GO TO LOOP; END;
        IF ACTION < 4 THEN DO;
          READ FILE(MASTER)KEY(INHERE.KEY)INTO(AREA);
          FIELD(ACTION) = CORRECTION;
          REWRITE FILE(MASTER)KEY(INHERE.KEY)FROM(AREA);
          GO TO LOOP; END;
FINISH: END EX110;
```

19.4
ADDITIONAL INPUT/OUTPUT SERVICES

PL/I provides a variety of services and messages which facilitate program debugging and aid in program design.

1. The DISPLAY statement is a way of receiving output quickly and conveniently on the console. Its basic form is

   DISPLAY (*expression*);

   This may be extended by means of the REPLY option as follows:

   DISPLAY (*expression*) REPLY (*v*1);

   where *v*1 is a character-string variable. When this form is used, the expression is evaluated and written, at which point the program stops. It will not resume until somebody types in a response on the console. That response is stored in *v*1 and the program resumes.
2. The keyword TIME refers to a predefined location in which the current time is available for output or other uses. For example, the statement

   PUT LIST (TIME);

   will print the current time as a nine-character string consisting of the hour, minute, second, and millisecond.
3. The keyword DATE operates like TIME. When used, it delivers a six-digit field consisting of the year, month, and day, in that order.
4. The ONFILE function delivers the name of the file involved in the most recently executed input/output statement. Its basic usage is in statements such as

   *v*1 = ONFILE;

   where *v*1 is a character string of sufficient length to accommodate the file name.
5. The number of data items transmitted in the last GET or PUT statement involving a specified stream file is stored in COUNT. Typical usage is

   *v*1 = COUNT(*filename*);

6. We can make reference to a PRINT file's line counter by writing a statement such as

   *v*1 = LINENO (*filename*);

   in which case *v*1 will contain the line number that was last written for *filename*.

PROBLEMS

1. A deck of cards contains the elements of a 6×8 matrix of four-digit integers punched one column per card in a manner suitable for list-directed transmission. Compute the transpose of this matrix (that is, make $A_{ij} = A_{ji}$) and punch out the result one column per card.
2. A deck of cards contains two fields per card punched in a manner suitable for data-directed transmission. Each card contains meas-

urements of two related variables x and y in the form XXX.XX. The entire file is to be used to find the straight line that *best* represents all the data—that is, the values of a and b in the function

$$y = a + bx$$

are to be determined. This is accomplished by solving the normal equations

$$Na + \Sigma xb = \Sigma y$$
$$\Sigma xa + \Sigma x^2 b = \Sigma xy$$

In these equations N is the number of pairs of values for x and y; Σx is the sum of all the x's; Σy is the sum of all the y's; Σx^2 is the sum of all the x's squared; and Σxy is the sum of all the products of related values (on the same card) of x and y. Write a program to read the cards, compute a and b, and display the results on the printer.

3. Produce a table, including a title and column headings, of the squares, cubes, and square roots of the integers from 1 through 100.
4. Produce a graph via the printer of the function Y = SIN(X), letting Y be measured horizontally and X vertically on the output form. Let the function complete at least one full cycle, incrementing X in steps of $\pi/36$, and let the graph swing between positions 10 and 110 on the output form. Use an appropriate plotting character, such as the asterisk or the period.
5. A library has a punched-card file for all its books, two cards being used for each book according to the following format:

Positions	*Contents*
1-10	Local classification number
11	1
12-18	Library of Congress number
19-26	Date of acquisition
27-32	Copyright date
33-35	Publisher code
36-40	Location code
41-45	Purchase price (point not punched)
46-47	Number of copies
48-80	Author
81-90	Local classification number
91	2
92-160	Title

Write a program to convert this card file to magnetic tape, blocking every five logical records into a physical record. Check to make sure that there are two and only two cards for each book and that the first has 1 in column 11 and the second has 2 in column 11. If any irregularity is detected, pause to make the necessary adjustments and then continue with the operation.

6. Write a program to read the tape file of Problem 5 and to divide

it into two separate files. The first file is a tape file consisting of all books whose copyright date is prior to January 1, 1945. Block every five logical records into a physical record. The second file, containing the remaining records, is a disk file consisting of 160-character records. The five-digit keys are computed from the ten-digit local classification numbers.

7. Using the tape file produced in problem 6, compute and print a frequency distribution showing the number of books acquired and the money spent for them in each year since the founding of the library in 1907.
8. A deck of cards is punched for use in updating the disk file produced in Problem 6. It has the following characteristics:
   (a) For new titles added to the library, two cards in the format shown in Problem 5 are punched. Check for copyright dates prior to January 1, 1945, and do not add any of these to the file; instead, notify the operator.
   (b) For corrections to the quantities, a single card is punched in the following format:

Position	*Contents*
1-10	Local classification number
11	3
12-18	Library of Congress number
19-45	Not used
46-47	Number of copies
48-80	Not used

   If the corrected number of copies is 0, the record is to be deleted from the file. Write a program to update the disk file according to the above considerations.
9. Using the updated disk file of the previous example, produce a printed report showing all titles acquired in the last full calendar year. On each line, show the local classification number, location code, number of copies, and title. Print the title of the report on the third line of the first page, column headings on the sixth line of each page, and the body lines single-spaced from the ninth to the third from the last line on each page. [*Hint:* In searching the file, let the keys be consecutive numbers from 1 through 99999, and use ON KEY to ignore keys for which no corresponding record exists.]
10. Text is prepared on punched cards starting in column 1 and continuing along until the last word FINIS is reached. Sentences are ended with a period followed by a blank, and there is one blank between words. No other punctuation is used and the period has no other meaning. Words continue through column 80 onto column 1 of the next card, etc. Write a program that prints each sentence on a separate line.

11. Using the input in Problem 10, write a program that creates a tape file in which each sentence is preceded by its respective sentence number. No sentence is longer than 100 characters and the number of sentences will not exceed 10,000.
12. Using the file generated in Problem 11, write a program that replaces specific sentences in that file. The sentences to be replaced are entered on cards sorted by sentence number. Each sentence starts in column 7 of a new card, with the sentence number in columns 1-5 of that card. If necessary, that sentence is continued onto a new card starting in column 1. The text continues right through column 80, as in Problem 10.
13. Using the file from Problem 11, determine and print (a) average number of words in each sentence, (b) sentence number with the greatest number of words, and the value of that number, (c) number of words beginning with each letter of the alphabet.
14. Using the file from Problem 11, determine the number of unique words and print each word, the number of times it occurs, and the sentence number for each occurrence. Asssume there is sufficient memory for all tables but not for the text itself.

Chapter 20

Additional PL/I Features

The previous two chapters, massive as they are, have provided only an introduction to a rather basic subset of PL/I's scope and capabilities. Since we cannot hope even to touch upon all of the remaining language features, we shall address ourselves in the chapter to several areas which expand the PL/I programmer's flexibility in ways which parallel, more or less, those associated with FORTRAN and COBOL.

20.1 ORGANIZATION OF PL/I PROGRAMS

Up to now we have been dealing with PL/I programs as single procedures. The only organizational component within these programs has been the DO jump or DO loop, with its own END statement. This organizational capability can be expanded substantially by means of a variety of subprograms.

20.1.1 Function Procedures

The function procedure provides the programmer with a way of adding his own built-in functions to those in PL/I's standard library. These functions, each with its characteristic name, may be available for a single

program or they can be made a permanent part of the language for a particular installation. A function subprogram is organized basically as follows:

```
name: PROCEDURE (parm1, parm2, . . . .);
statement;


.
statement;
RETURN (expression);
END name;
```

where *name* is a label which then serves as the "keyword" by which the function is *invoked*, *parm*1, *parm*2, ... represent a list of *parameters* (or *dummy arguments*) which help define the operations performed by the function. Storage is not actually reserved under the names. When the function is invoked, the programmer supplies a list of arguments whose number and attributes are consistent with those of the parameter list.

EXAMPLE 111

A library keeps the following keypunched record for each book out on circulation:

Column(s)	*Contents*	*Format*
1–7	Call number	Characters
10–11	Allowed time out, days (07, 14, or 28)	XX
15–20	Date taken out, mo., day, year	XXXXXX
22–80	Other identification	Characters

Treating this set of cards as a stream file, read in today's date and print the following for each overdue book: call number, identification and number of days overdue. All calendar days count, and to make things simple, forget about leap year.

Solution Procedure

We shall set up a separate function to calculate the number of overdue days. The algorithm we shall use refers both dates to 01/01/01, the first day of the century. Thus if M_c, D_c, and Y_c are the current values and M_i, D_i, and Y_i are the initial values (the date a book was taken out),

$$\text{Days} = 365(Y_c - 1) + \sum_{j=1}^{j=M_c} N_j + D_c - 365(Y_i) + \sum_{j=M_i}^{j=12} N_j - D_i - N_c$$

where N_i is the number of days in month i, obtained from a table. The arguments given to the function will be the two dates and the allowed circulation time.

Solution Program

```
EX111:    PROCEDURE OPTIONS(MAIN);
          DECLARE BOOK CHARACTER(7), IDENT CHARACTER(59), (YC,MC,-
          DC,YI,MI,DI,ALLOW)
            FIXED(2), OVER FIXED(4);
          ON ENDFILE GO TO FINISH;
          GET EDIT (MC,DC,YC) (3 F(2), SKIP);
BRING:    GET EDIT (BOOK,ALLOW,MI,DI,YI,IDENT) (A(7),X(2),F(2),X(3),3F-
          (2),X(1),A(59));
          OVER = OVERDUE (MC,DC,YC,MI,DI,YI,ALLOW);
          IF OVER <= 0 THEN GO TO BRING;
          PUT SKIP EDIT (BOOK,IDENT,OVER) (X(5),A(7),A(5),A(59),X(5),F(4));
          GO TO BRING;
OVRDUE:   PROCEDURE (M2,D2,Y2,M1,D1,Y1,X);
          DECLARE (MANY(12),M2,D2,Y2,M1,D1,Y1,X) FIXED(2), DAYS FIX-
          ED(4) INITIAL(0);
          MANY(1), MANY(3), MANY(5), MANY(7), MANY(8), MANY(10),
          MANY(12) = 31;
          MANY(4), MANY(6), MANY(9), MANY(11) = 30; MANY(2) =
          28;
            DO I = 1 TO M2;
            DAYS = DAYS + MANY (I);
            END;
          DO I = M1 TO 12;
          DAYS = DAYS + MANY(I);
          END;
          DAYS = DAYS + 365*(Y2 - 1) - 365*(Y1) + D2 - D1 - MANY(M2);
          RETURN (DAYS-X);
          END OVRDUE;
FINISH:   END EX111;
```

Function procedures may be made more flexible by having several points of entry and/or exit. The basic structure is outlined below:

```
name1: PROCEDURE (param list);
          statement;

          .

name2: ENTRY (param list);
          statement;

          .

name3: RETURN (expression);
```

*name*4: RETURN (*expression*);

.

END *name*1;

The parameter lists may or may not be identical; the same holds true for the expressions in the RETURN statements.

20.1.2 Subroutines

PL/I subroutines are built like function procedures, the main difference being that they can return more than one value. To do this, the programmer supplies the destinations of the results as part of the argument list. Correspondingly, it is not necessary to attach an expression to the RETURN statement (in fact, it is illegal).

A subroutine is invoked by the statement

CALL *name* (*argument list*);

EXAMPLE 112

To examine the basic organization of a subroutine, repeat example 111 with the overdue calculation being done as a subroutine. Return the number of days overdue (zero if not overdue) and a code of 0 if the book is not overdue, 1 if overdue less than a week, 2 if overdue 1–2 weeks, and 3 if overdue more than two weeks. Print an asterisk next to the number of days overdue if the code is 2 and two asterisks if the code is 3.

Solution Program

```
EX112:  PROCEDURE OPTIONS (MAIN);
        DECLARE BOOK CHAR(7), IDENT CHAR(59), (YC,MC,DC,YI,MI,DI,
        ALLOW)
          FIXED(2), OVER FIXED(4), CODE FIXED(1), ATTACH CHAR(2);
        ON ENDFILE GO TO FINISH;
        GET EDIT (MC,DC,YC) (3F(2),SKIP);
BRING:  GET EDIT (BOOK,ALLOW,MI,DI,YI,IDENT) (A(7),X(2),F(2),X(3),
        3F(2),X(1),A(59));
        CALL OVRDUE (MC,DC,YC,MI,DI,YI,ALLOW,OVER,CODE);
        IF CODE = 0 THEN GO TO BRING;
        IF CODE = 1 THEN ATTACH = ' ';
        IF CODE = 2 THEN ATTACH = '* ';
        IF CODE = 3 THEN ATTACH = '**';
        PUT SKIP EDIT (BOOK,IDENT,OVER,ATTACH) (X(5),A(7),X(5),A(59),
        X(5),F(4),A(2));
        GO TO BRING;
```

```
OVRDUE: PROCEDURE (M2,D2,Y2,M1,D1,Y1,X,Y,Z);
        DECLARE (MANY(12),M2,D2,Y2,M1,D1,Y1,X) FIXED(2), Y FIXED(4)
          INITIAL(0), Z FIXED(1);
        MANY = 31; MANY(4), MANY(6), MANY(9), MANY(11) = 30;
          MANY(2) = 28;
          DO I = 1 TO M2;
          DAYS = DAYS + MANY(I);
          END;
        DO I = M1 TO 12;
        DAYS = DAYS + MANY(I);
        END
        DAYS = DAYS + 365*(Y2 - 1) - 365*(Y1) + D2 - D1 - MANY(M2);
        DAYS = DAYS - X;
        IF Y <= 0 THEN Y = 0; IF Y = 0 THEN Z = 0; IF Y < 7 THEN Z = 1;
        IF Y < 14 THEN Z = 2; ELSE Z = 3; RETURN; END OVRDUE;
        END EX112;
```

20.1.3
Relations Among Procedures

Thus far we have shown function and subroutine procedures embedded in a main program. These are said to be *internal* procedures. As such, they must be fully contained in the main program. Internal procedures may be nested within each other according to a set of rules exactly like those governing DO loops (Sec. 18.7.3). That is, internal procedures may be nested to any level, but there can be no overlap; a procedure must be completely contained within another. Internal procedures may be invoked only by procedures at one level higher. We can illustrate this with the following schematic program layout:

```
P1: PROCEDURE OPTIONS (MAIN);

      .

   P2: PROCEDURE;

      .

      P3: PROCEDURE;

         .

      END P3;
```

```
END P2;

END P1;
```

In this situation P1 may invoke P2 but not P3. Similarly, P2 may invoke only P3. P3 may not invoke anything. If we would want P1 to call P3, the way to do it would be to have an entry point in P2 where the instruction is simply a call to P3.

P1 in this layout is an *external* procedure. A program may have several external procedures (one of which must be the main procedure). External procedures may be compiled separately and may each contain internal procedures on their own. An external procedure may not invoke an internal procedure contained elsewhere. Instead, it must call the containing external procedure.

PL/I's external procedures have some rather special properties with regard to the use of storage. The fact that a procedure is external means, among other things, that the variable names in that procedure are independent of those in other external procedures. Thus if the name B were to appear in two external procedures of a given program, PL/I would ordinarily treat them as two different variables. In fact, unless special precautions are taken, the B in the procedure that is currently inactive would have been wiped out. This is not true for internal procedures. A name used in an external procedure is generally *known* to all procedures contained therein, even if that name is not given in an argument list. For example, in the following program

```
EXT: PROCEDURE OPTIONS (MAIN);
     DECLARE (A,B,C) FIXED(5,3);
     GET LIST (A,B);
     CALL IN;
     PUT PAGE DATA (A,B,C);
       IN: PROCEDURE;
         C = 3*SIN (A+B);
         RETURN;
         END IN;
     END EXT;
```

the C that is printed will indeed have a value equivalent to 3 sin (A+B) even though the procedure IN has no argument list. Had we written the same program as

```
EXT: PROCEDURE OPTIONS (MAIN);
     DECLARE (A,B,C) FIXED(5,3);
```

```
        GET LIST (A,B);
        CALL EXT2;
        PUT PAGE DATA (A,B,C);
        END EXT;
EXT2:   PROCEDURE;
        C = 3*SIN (A+B);
        RETURN;
        END EXT2;
```

PL/I would not recognize the names. In order for such recognition to occur, a variable must be declared with the EXTERNAL (abbreviated EXT) attribute. This instructs PL/I to refer to that particular location each time that name is used on another procedure. The extent over which a particular name is associated with a given location is called the *scope* of that name.

EXAMPLE 113

Show the scope of the names in the following program organization:

```
OUT1: PROCEDURE OPTIONS (MAIN);
   L1: DECLARE ((X1,X2)EXTERNAL,Y) FIXED(4);
       statements
   L2: CALL IN1;
   L3: CALL OUT2;
   L4: CALL IN2;
       statements
         IN1: PROCEDURE;
          L5: DECLARE (X2,Y) FIXED(4);
              statements
         IN2: PROCEDURE;
          L6: DECLARE Y FIXED(4), X1 FLOAT(7);
              statements
              END IN1;
      statements
END OUT1;
OUT2: PROCEDURE;
   L7: DECLARE (X1,X2) FIXED (4) EXTERNAL, Y FLOAT(4);
       statements
       CALL IN3;
       statements
         IN3: PROCEDURE;
          L8: DECLARE (X1,X2,Y) FLOAT(4);
              statements
              END IN3;
              statements
END OUT2;
```

TABLE 20.1 USE OF NAMES IN EXAMPLE 113

Statement Label	Name	Attributes	Scope
OUT1	OUT1	ENTRY, EXTERNAL	Entire Program
L1	X1.X2	FIXED, EXTERNAL	All of OUT1 except for IN2, all of OUT2 except for IN3
L1	Y	FIXED, INTERNAL	OUT1 except for IN1 and IN2
IN1	IN1	ENTRY, INTERNAL	All of OUT1
L5	X2.Y	FIXED, INTERNAL	All of IN1
L6	Y	FIXED, INTERNAL	All of IN2
L6	X1	FLOAT, INTERNAL	All of IN2
OUT2	OUT2	ENTRY, EXTERNAL	Entire Program
L7	X1.X2	FIXED, EXTERNAL	Same X1 and X2 as in L1
L7	Y	FLOAT, INTERNAL	All of OUT2 except for IN3
IN3	IN3	ENTRY, INTERNAL	All of OUT2
L8	X1.X2.Y	FLOAT, INTERNAL	All of IN3

Solution

The scope of each name, and each reference thereto, is summarized in Table 20.1.

20.1.4 BEGIN Blocks

The BEGIN block, though not a separate procedure, is an organizational component that provides additional flexibility. It is executed in line (where it appears; it is not invoked like a function or subroutine). Operationally, the BEGIN block works like a DO group. That is, it can provide one of the alternatives of an IF statement and will be executed or bypassed as a unit. Such blocks are constructed as follows:

```
BEGIN;

.

END;
```

The basic difference between a DO group and a BEGIN block lies in the latter's ability to affect the scope of names in a program. A BEGIN block acts exactly like an internal procedure in that respect. If a variable name appearing in the containing procedure also appears in a contained BEGIN block without its being declared there, PL/I will interpret it to be the same variable. If, on the other hand, that variable name appears in a separate declaration in a BEGIN block, it is treated as a separate and distinct variable.

20.2 ADDITIONAL FACILITIES FOR INTERRUPT HANDLING

We have seen the usefulness of the ON statement for a variety of input/output conditions. This is but one side of PL/1's capabilities for imposing programmer-defined actions for a number of specific types of interrupts.

ON (condition) may be followed by a single statement or by a BEGIN block. When followed by a single statement, normally a GO TO statement, subsequent program control depends upon that statement; when followed by a BEGIN block, program control is returned to the statement following the one in which the error or special condition arose once the statements of the BEGIN block have been executed.

Unless some specification to the contrary is made, that action specification established by an ON statement remains in effect from that point on. Specifications to the contrary include (1) execution of another ON associated with the same condition and designating other action, (2) transfer of control to a block where either another ON or no ON is in effect, and (3) execution of the REVERT statement. The REVERT statement (REVERT followed by a condition, for example, REVERT ZERODIVIDE) nullifies the effect within the program block in which it is executed of the most recent ON statement previously executed for that condition.

There are 19 conditions that may be used as operands in ON, REVERT and SIGNAL statements. Seven of these have been discussed in the previous chapter (Sec. 19.3.1) and two are beyond our scope in this text. The others are as follows:

CONVERSION (CONV)	UNDERFLOW (UFL)	CONDITION
FIXEDOVERFLOW (FOFL)	ZERODIVIDE (ZDIV)	FINISH
OVERFLOW (OFL)	SUBSCRIPTRANGE (SUBRG)	CHECK
SIZE		

The SIGNAL statement (SIGNAL followed by a condition, for example, SIGNAL ZERODIVIDE) allows the programmer to simulate the occurrence of a condition and is particularly useful in checking out special procedures. SIGNAL ZERODIVIDE, for example, causes the same effect as ON ZERODIVIDE when an attempt to divide by zero is made.

CONVERSION is a condition that arises when an attempt is made to convert a character string field containing improper characters to other types of fields.

FIXEDOVERFLOW is a condition that arises when the result of a fixed-point arithmetic operation exceeds the maximum field width allowed for such a result.

OVERFLOW is a condition that arises when the exponent of a floating-point number exceeds its maximum permitted value.

SIZE is a condition that arises when a conversion between data types or between differing bases, scales, or precisions results in the loss of high-order digits. SIZE depends on the declared size of a field; FIXEDOVERFLOW depends on the field size allowed by the computer.

UNDERFLOW is a condition that arises when the exponent of a floating-point number is smaller than its minimum permitted value.

ZERODIVIDE is a condition that arises when an attempt is made to divide by zero, with respect to either fixed-point or floating-point division.

SUBSCRIPTRANGE is a condition that arises when a subscript is evaluated and is found to lie outside its specified bounds.

CHECK (identifier list) is a condition that arises when any one of the identifiers may be entry names, statement labels, or variable names.

CONDITION (identifier) is a condition that arises when a SIGNAL statement having the same identifier is executed. This condition is programmer-named and may be used for any purpose he wishes.

FINISH is a condition that arises immediately before the main procedure terminates by executing a STOP, RETURN, END, or EXIT statement.

In addition to conditions being named in ON statements, they may be applied or ignored on a selective basis; in some cases this selection is mandatory. Any condition whose occurrence causes an interrupt is said to be *enabled*. Enabling of UNDERFLOW, OVERFLOW, ZERODIVIDE, FIXEDOVERFLOW, and CONVERSION is provided automatically by PL/I; enabling of other conditions must be performed by the programmer. Enabling a condition is specified by enclosing it in parentheses and by placing it immediately before a statement, including its label, and separating it from the statement by a colon. If the statement is a PROCEDURE or BEGIN statement, the condition is enabled for the whole block; if the statement is otherwise, the condition is enabled for the statement only. For example,

```
(SIZE): ROOTS: PROCEDURE;
```

enables the SIZE condition for the entire procedure block whose entry name is ROOTS. Again,

```
(SIZE, SUBSCRIPTRANGE): A(K,I,J) = B**C;
```

enables both SIZE and SUBSCRIPTRANGE for the given assignment statement only.

Conditions may be *disabled* as well as enabled. Disabling is specified by writing NO immediately preceding (with no intervening blanks) the condition name and using the above prefix notation. For example,

```
(NOZERODIVIDE): LOOP: Y = X/Z;
```

disables the ZERODIVIDE condition during the assignment statement whose label is LOOP.

The ON statement, in addition to naming a condition and specifying the action to be taken, may include the word SNAP after the condition. This indicates that a trace is to be executed and listed upon the occurrence of the specified condition. An example of this usage is

```
ON SIZE SNAP GO TO RECOVER;
```

Also, the condition specified in an ON statement may be followed by the word SYSTEM. This designates that the standard system action is to be taken on the occurrence of the specified condition. An example of this usage is

```
ON OVERFLOW SYSTEM;
```

Standard system action consists of printing a comment and terminating for the following conditions: CONVERSION, OVERFLOW, SIZE, ZERODIVIDE, ENDFILE, TRANSMIT, UNDEFINEDFILE, KEY, RECORD, and SUBSCRIPTRANGE. Such action consists of printing a comment and continuing for FIXEDOVERFLOW, UNDERFLOW, NAME and CONDITION. For ENDPAGE, standard system action is to start a new page; for FINISH, to terminate the job. Standard system action for the CHECK condition is to write the identifier on a debugging file; if the identifier is a field name, the new value of the field is also written on the debugging file.

20.3 DYNAMIC ALLOCATION OF STORAGE

Because the memory of any computer is limited in capacity, the efficient use of this memory during the execution of a program is often important. The simple static process of space allocation used by many compilers—the assignment of a distinct region for each distinct variable used in the source program—may be wasteful. Multiple use of a memory area for different data during program execution can reduce the total amount of memory space required.

Provisions are included within PL/I to give the programmer virtually any degree of control over the allocation of memory for the data variables in a program. On the other hand, the entire problem of allocation can be ignored completely by the programmer if memory economization is of little significance in his situation, and a reasonably efficient use of memory space is usually still obtained automatically.

Memory space is said to be *allocated* for a variable when a certain region of memory is associated with the variable. Allocation for a given variable may take place *statically*, before execution of the program, or *dynamically*, during execution. Memory space may be allocated dynamically for a variable and subsequently *released*. Thus this region is freed for possible use in later allocations. If space has been allocated for a variable and not subsequently

released, the variable is said to be *in an allocated state.* Whenever reference to a variable is made in a PL/I statement, the variable must be in an allocated state.

The manner in which memory space is allocated, when not carried out by default, is by means of special attributes declared to be associated with variables. These are three such attributes: STATIC, AUTOMATIC, and CONTROLLED.

Space for a variable declared to have the STATIC attribute is allocated before execution of the program and is never released during execution. Such a variable may also have either the EXTERNAL attribute, so that it can be used for communication between different external procedures, or the INTERNAL attribute, in which case its definition applies only to the block to which the declaration is internal but excluding all contained blocks to which another declaration of the same identifier is internal. An EXTERNAL variable for which the allocation attribute is not specified is assigned the STATIC attribute by default.

Space for a variable declared to have the AUTOMATIC attribute is allocated dynamically whenever the block containing the variable is activated. Such space remains allocated until the block is terminated, at which time the space is released. Because the dynamic allocation of memory space for variables with the AUTOMATIC attribute is controlled on a block-by-block basis, variables with this attribute must also have the INTERNAL attribute. In addition, an INTERNAL variable for which the allocation attribute is not specified is assigned the AUTOMATIC attribute by default.

Space for a variable declared to have the CONTROLLED attribute, except for the based variables encountered in the previous chapter, is allocated by using the special ALLOCATE statement and released using the special FREE statement. Determination of the points at which memory space is either allocated or released, therefore, is made by the programmer. A controlled variable may be EXTERNAL or INTERNAL.

In the following example, a skeletal program structure is used to illustrate memory allocation.

EXAMPLE OF MEMORY ALLOCATION

```
A: PROCEDURE;
   DECLARE X STATIC;


   .
B: PROCEDURE;
   DECLARE Y(100) CONTROLLED,
     Z CHARACTER (1000);
```

```
    ALLOCATE Y;

    .
    FREE Y;

    .
    C: BEGIN;
       DECLARE Z(100);

    .
    END C;

    .
RETURN;

    .
END B;

    .
END A;
```

Assume for the purposes of the following discussion that procedure A is terminated by END A, the procedure B is terminated by RETURN, and that procedure C is terminated by END C. Then the following statements can be made:

1. Space for the STATIC variable X is allocated before execution and is never released.
2. The character string variable Z is AUTOMATIC by default. Space for this Z is allocated on entry to procedure B and is released on execution of the RETURN statement.
3. The array variable Z is AUTOMATIC by default. Space is allocated for this Z at the beginning of execution of block C and is released at END C.
4. Space for the CONTROLLED variable y is allocated on execution of the ALLOCATE statement and is released on execution of the FREE statement. After execution of the FREE statement, the variable Y is not used, but the character string variable Z may be used.

The FREE statement illustrated above may become slightly more complex by the inclusion of multiple variables separated by commas; the ALLOCATE statement, in contrast, has numerous alternatives.

The ALLOCATE statement may specify multiple variables separated by commas, and each of these may have the identifier preceded by a level number and followed by a dimension attribute and/or a BIT, CHARACTER, or INITIAL attribute. Alternatively, based variable identifiers may be followed by SET and/or IN clauses. When a SET clause is used, this keyword is followed by the name of a pointer variable in parentheses. When the IN clause is used, the keyword is followed by the name of an area variable in parentheses. Pointer variables, when not declared contextually, are specified in the DECLARE statement using the POINTER attribute. Area variables are declared using the AREA attribute, and are used to define an area of memory that may be used for collecting and referring to based variables.

The POINTER attribute is used for the explicit declaration of an identifier as the name of a pointer variable; for example:

```
DECLARE P POINTER STATIC;
```

The pointer PT1 is declared contextually by its appearance with the CONTROLLED attribute, as in the statement:

```
DECLARE VALUES CONTROLLED (PT1);
```

Pointers are also declared contextually by their appearance in the SET option of ALLOCATE, READ, and LOCATE statements, as in the following:

```
ALLOCATE VALUES SET (PT3);
```

The CELL attribute allows use of specific memory space by two or more variables, only one of which may be active (contain a value) at a time. Each of the alternate variables may have any of the attributes that a structure component may have.

If a FREE statement names a variable for which no space has been allocated at the time the FREE statement is executed, this condition is recognized as an error. In order to be able to test whether space has been allocated for a nonbased CONTROLLED variable (including structures and unsubscripted arrays), a built-in function is available. ALLOCATION (X) is set to '1'B if memory space has been allocated to X and is set to '0'B otherwise.

For certain variables in any sizable program, it makes no difference whether memory space is efficiently assigned, and it is wasteful to have the compiler attempt to optimize such assignment. If these variables are major structures, arrays, and variables not contained in structures or arrays, that is, if they are level 1 variables, the SECONDARY attribute may be used to specify that efficient storage is not required.

These features, together with several others, endow PL/I with a powerful repertoire of facilities for list processing. This particular aspect of the language is explored in Refs. 1 and 2 listed at the end of this chapter.

20.4 ADDITIONAL STORAGE-HANDLING FEATURES

PL/I has the widest array of facilities for assigning and reassigning storage. Among these are included a number of features for attaching several names to the same locations and for setting word boundaries.

20.4.1 The DEFINED and POSITION Attributes

The DEFINED attribute (abbreviated DEF) specifies that data are to occupy the same memory locations as those assigned to other data. The DEFINED keyword is followed by the name of a variable, array, or structure, optionally followed by relevant subscripts. Brief examples, which also contain the POSITION attribute follow:

```
DECLARE LIST CHARACTER(40),
        ALIST CHARACTER(10) DEFINED LIST;
```

ALIST corresponds to the first ten characters of LIST.

```
DECLARE LIST CHARACTER(40),
        BLIST CHARACTER(20) DEFINED LIST POSITION(21);
```

BLIST corresponds to the twenty-first through fortieth characters of LIST, the POSITION attribute (abbreviated POS) serving to designate the starting point of the definition.

20.4.2 The *i*SUB Feature

A part of an array may be assigned a separate array name through the use of the *i*SUB feature. *i*SUB (where *i* is a decimal integer) defines the type of correspondence between a subscript value in the defined array and one in the base array. The *i* indicates which dimension of the base array is to be used for this function. (If the base array is one-dimensional, *i* of course can only have a value of 1.)

As an example, suppose we have a base array X declared as

```
DECLARE X(24) FIXED(2);
```

then, if we said

```
DECLARE Y(4) FIXED(2) DEFINED X(6*1SUB);
```

we would be attaching the duplicate names Y(1), Y(2), Y(3), and Y(4) to the respective locations named X(6), X(12), X(18), and X(24). This is a very flexible thing. For instance, with X as before, we could say

```
DECLARE Z(2,3) FIXED(2) DEFINED X(4*|SUB-1);
```

in which case the following duplicate names would be assigned:

X *element*		*Is Also Named*
X(3)	(X(4*1-1))	Z(1,1)
X(7)	(X(4*2-1))	Z(1,2)
X(11)	(X4*3-1))	Z(1,3)
X(15)	etc.	Z(2,1)
X(19)		Z(2,2)
X(23)		Z(2,3)

20.4.3 The ALIGNED, PACKED, and LIKE Attributes

The ALIGNED and PACKED attributes are used to specify in memory the arrangement of string or numeric field data elements within data aggregates. Those attributes may be specified for names of major structures and names of arrays that are not themselves part of a structure.

PACKED specifies that each string or numeric field element is packed in memory contiguous with the string or numeric field elements that surround it; ALIGNED specifies that each string data element within the aggregate may start at a memory boundary, such boundary (halfword, word, doubleword) being a function of the individual PL/I compiler.

In the example which follows, all elements of A, an array of structures, occupy a continuous area of memory. Each element of the array E starts at a compiler-defined memory boundary. There may be unused space between the elements of the latter array.

```
DECLARE
  1 A (10) PACKED, 2 B BIT
  (200), 2 C BIT (500), 2 D BIT
  (300), 2 E (10,15) ALIGNED BIT (15);
```

The LIKE attribute specifies that the name being declared is given the same structuring as the name following the keyword LIKE. The following two statements, for example, are equivalent to each other:

```
DECLARE 1 A EXTERNAL, 2 (B,C,D),
        1 E LIKE A;
DECLARE 1 A EXTERNAL, 2 (B,C,D),
        1 E, 2 (B,C,D);
```

The structure name that follows the LIKE keyword may be unqualified or qualified, but may not be subscripted; in addition, it must be defined in the block containing the LIKE keyword.

PROBLEMS

1. The American Shmichik Company has the following schedule of rates for its product:

Number	*Unit Price*	*Number*	*Unit Price*
< 100	$4.47	500–999	$4.15
100–499	4.32	1000+	$3.95

Sales orders are punched as follows:

Columns	*Contents*	*Format*
1–6	Date, mmddyy	XXXXXX
7–10	Customer Number	XXXX
17–20	Number of units ordered	XXXX
21–80	Customer Identification	characters

Write a program that produces an invoice for each order read in. The format should be as shown below:

AMERICAN SHMICHIK COMPANY
INVOICE FOR
customer's identification
CUSTOMER NO. XXXX
Date mm/dd/yy

XXXX SHMICHIKS---------- $XXXXXX.XX

2. Modify the program in Problem 1 to accommodate a pricing policy which applies the following discounts:
   (a) Shmichiks ordered in February or October in quantities exceeding 499 are discounted 5%.
   (b) Orders of 1,000 or more placed in March are discounted 7%.
   (c) All orders placed in January are discounted 3%.
3. Assume that there exists a sequential file called SUMMARY in which each record appears as follows:

Positions	*Contents*	*Format*
1–4	Customer Number	XXXX
5–64	Customer Identification	characters
65–71	Total number of units ordered	XXXXXXX
72–80	Total Sales	$XXXXXX_{(.)}XX$

This file is closed out at the end of each year and is maintained in ascending customer number order. The input cards are presorted the same way. Modify the program of Problem 2 so that, in addition to doing what it did there, it adds a customer's current order to his record, updating the total number and total sales. There may be more than one order card for a given customer. Declare the SUMMARY file any way you want, as long as it is sequential.

4. Repeat Problem 3 with SUMMARY as a direct file and the input cards in any order.
5. The specific heat for certain kinds of compounds can be expressed by the general formula

$$Cp = a_0 + a_1T + a_2T^2$$

where T is the temperature in degrees Réaumur (Fahrenheit + 460) at which Cp is to be determined and a_0, a_1, and a_2 are characteristic constants for a particular compound. A chemical company has evaluated such constants for a series of compounds and wishes to publish tables of specific heats from 460° R through 800° R at 1° increments. The data for each compound are punched as follows:

Columns	*Contents*	*Format*
1-5	Compound Number	XXXXX
6-20	Compound Name	characters
21-26	a_0	$XX_{(.)}XXXX$
31-36	a_1	$XX_{(.)}XXXX$
41-46	a_2	$XX_{(.)}XXXX$

Write a program to produce these tables, printing no more than 50 values to a page. Head each page as indicated:

SPECIFIC HEAT DATA FOR
(compound name, compound number)
TEMPERATURE SPECIFIC HEAT

6. Repeat Problem 5, modifying the program to accept a separate temperature range for each compound. Lowest temperature, in degrees Fahrenheit, appears in columns 51-53 and the highest temperature, also in degrees Fahrenheit, is punched in 61-63.
7. Show the order in which the statements are executed in Example 113.
8. We have a 10 × 10 array of three-digit integers called REF.
   (a) Declare a one-dimensional array DIAG consisting of the diagonal of array REF.
   (b) Declare a one-dimensional array ODDC consisting of those of REF's elements in row 4 with odd columns numbers.
   (c) Declare a two-dimensional square array OTHER consisting of those elements of REF with odd row numbers and even columns numbers.

REFERENCES

1. Pollack, S. V., and T. D. Sterling, *A Guide to PL/1*. New York: Holt, Rinehart and Winston, Inc., 1969.
2. Lawson, H. W., Jr., "*PL/1 List Processing*," *Comm. of the ACM*, Vol. 10, No. 6 (June 1967).

Appendix A

The ALGOL Language

ALGOL (ALGOrithmic Language) has achieved widespread use as a language for the description of algorithmic processes and for international communication among those whose prime interests have been in the analytical aspects of computation. It has been used as a programming language as well, but has not received broad acceptance.

Information on the System/360 implementation of ALGOL has been relegated to this appendix because interest in and usage of this language are not expected to become sizable. The major reason is that those for whom FORTRAN is not entirely adequate can turn to PL/I and have not only an improved algorithmic language but a general-purpose language as well.

The brief description of the ALGOL language that follows is given in five major parts: basic definitions, expressions, statements, declarations, and input/output procedures.

A.1 BASIC DEFINITIONS

A.1.1 Symbols

ALGOL contains four types of *basic symbols: letter* (A–Z), *digit* (0–9), *logical value* ('TRUE', 'FALSE'), and *delimiter* (defined below). Single quotation marks bound all keywords.

TABLE A.1 ALGOL DELIMITERS

Arithmetic Operators	Relational Operators	Logical Operators	Sequential Operators
+	'LESS' or <	'EQUIV'	'GOTO'
−	'NOTGREATER' or <=	'IMPL'	'IF'
*	'EQUAL' or =	'OR' or |	'THEN'
/	'NOTEQUAL' or ¬=	'AND' or &	'ELSE'
'/'	'NOTLESS' or >=	'NOT' or ¬	'FOR'
'POWER' or **	'GREATER' or >		'DO'

Separators	Brackets	Declarators	Specificators
	(	'BOOLEAN'	'STRING'
	)	'INTEGER'	'LABEL'
. or :	(/	'REAL'	'VALUE'
,, or ;	/)	'ARRAY'	
:= or ,,= or ;=	'(' (left string quote)	'SWITCH'	
'STEP'	')' (right string quote)	'PROCEDURE'	
'UNTIL'	'BEGIN'	'CODE'	
'WHILE'	'END'		
'COMMENT'			

ALGOL's five types of delimiters are: *operator* (arithmetic, relational, logical sequential), *separator, bracket, declarator,* and *specificator.* These are summarized in Table A.1

A.1.2 Blanks and Comments

The ALGOL compiler ignores blanks inside keywords. For instance, 'BEGIN' and 'BE GI N' are tested the same. Comments (also ignored) may be introduced in any of these ways:

1. .,'COMMENT'----remarks----.,
2. 'BEGIN' 'COMMENT'---remarks---.,
3. 'END'---remarks---'END'
4. 'END'---remarks---.,
5. 'END'---remarks---'ELSE'

A.1.3 Identifiers

Identifiers in ALGOL are used to name variables, arrays, labels, switches and procedures. As such they must begin with a letter and may contain nothing but letters and digits. There is no maximum length, but identifiers having the same first six characters will cause ALGOL to refer to the same location. (Thus we can assign different names to the same item.)

A.1.4
Numbers

ALGOL numbers may be declared as 'INTEGER' or 'REAL' and are written as decimal quantities, signed or unsigned, with or without exponents. If a decimal point is shown, it must be followed by a digit (even if that digit is zero). Some examples appear in Table A.2.

TABLE A.2 ALGOL NUMBERS

Conventional Notation	ALGOL Notation	Conventional Notation	ALGOL Notation
0	0	-9.80×10^{2}	$-9.80'2$ or $-9.80'+2$
212	212	6×10^{-3}	$6'-3$
.671	.671 or +.671	$.007 \times 10^{-2}$	$.007'-2$
−0.027	−0.027	-10^{3}	$-'3$ or $-'+3$
+3.02	+3.02 or 3.02	10^{-5}	$'-5$
2.89×10^{4}	2.89'4 or +2.89'+4	10^{5}	$'+5$ or $'5$ or $+'+5$

'INTEGER' numbers are stored as fixed-point binary fullwords (31 bits plus sign); 'REAL' numbers are stored as floating-point fullwords or doublewords, as directed by the programmer.

A.1.5
Strings

1. A *proper string* is any sequence of characters not containing '(' or ')'.
2. An *open string* is a proper string bounded on the left by '(' and on the right by ')'. A series of open strings may form another open string.
3. A string is an open string enclosed by '(' and ')'.

Examples of strings:

Proper string: THESE 19 CHARACTERS (this is also an open string)
Open strings: '('THESE 19 CHARACTERS')'
THIS AND '('THESE 19 CHARACTERS')'
Strings: '(''('THESE 19 CHARACTERS')'')'
'('THIS AND '('THESE 19 CHARACTERS')'')'

A.1.6
Quantities and Scopes

Quantities are of five kinds: *simple variables, arrays, labels, switches,* and *procedures.* The *scope* of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid.

A.1.7
Expressions

There are three basic types of ALGOL expressions: Arithmetic, Boolean and designational. The constituents—logical values, numbers, variables, function designators, labels, switch designators, and elementary arithmetic, relational, logical, and sequential operators—are introduced below.

A.1.7.1
Variables

Variables are named by identifiers (Sec. A.1.3) and may be *simple* or *subscripted*. When it is of the latter type, its identifier is called an *array identifier* and its declaration includes a *subscript list*. For example a subscripted variable having 4 rows and 7 columns and the name Y would be declared as Y(/4,7/). ALGOL accepts as many as 16 dimensions.

A.1.7.2
Logical Values and Numbers

Logical values are 'TRUE' or 'FALSE'. Numbers have already been defined in Sec. A.1.4.

A.1.7.3
Function Designators

Function designators are analogous to expressions (or parts of expressions) in PL/I or FORTRAN which invoke functions (built-in or user-designed) which return single values. The name of the function in ALGOL is called the *procedure identifier*. Whenever an ALGOL function is invoked, the procedure identifier is accompanied by a list of appropriate *actual parameters* upon which the function operates. An actual parameter may be a string, a variable, a switch identifier, an expression or another procedure identifier. Actual parameters are separated by commas and the list is generally enclosed in parentheses.

ALGOL has ten *standard functions*, each with a unique procedure identifier. These are summarized in Table A.3.

A.1.7.4
Arithmetic Expressions

Combinations of the foregoing components (excluding non-numeric values) may be combined by arithmetic operators to form arithmetic expressions. These follow algebraic notations fairly closely and can be stated unambiguously by including parenthesized terms. The various components have standard names, as given below Table A.3:

TABLE A.3 STANDARD ALGOL FUNCTIONS

Example of Function Designator	Conventional Equivalent	Argument Type	Result
ABS(X)	$\lvert X \rvert$	'REAL' or 'INTEGER'	'REAL'
SQRT(X)	$\sqrt{X}$		
SIN(X)	sin X (X in radians)		in radians
COS(X)	cos X (X in radians)		
ARCTAN(X)	tan^{-1}X		
LN(X)	$\log_e X$		'REAL'
EXP(X)	e^X		
SIGN(X)	$+1.0, -1$ if $X > 0, X = 0$, or $X < 0$, respectively		'INTEGER'
ENTIER(X)	largest integer $\leq X$	'REAL'	'INTEGER'
LENGTH(C)	Number of characters in the open string between C's outermost string quotes	characters	'INTEGER'

Primary: an unsigned number, single valued variable, function designator or parenthesized arithmetic expression. Thus X, Y(/1,2/), LN(2*Y −3), 3.2′ −1, and (3 + 2*X) are primaries.

Factor: a primary, or a primary raised to a primary power. Using the primaries from before, X'POWER'(3+2*X) would be a factor. A factor can be used in place of the leading primary to form another factor. Thus, X'POWER' Y(/1,2/) is also a factor. Similarly, the primary X is also a factor.

Term: a factor, or two factors connected by * or /. Thus Y(/1,2/)*X is a term. The first part of a term may also be a term, so that

Y(/1,2/)*X/X'POWER'(3+2*X)

is a term.

Simple arithmetic expression: two terms connected by + or −. The first term may itself be a simple arithmetic expression. Thus, X+2*Y(/1,2/) is a simple arithmetic expression. These may be used as subscripts.

A.1.7.5 Boolean Expressions

Boolean expressions work down to single values of 'TRUE' or 'FALSE'. As such, they may be constructed from a variety of components:

Relation: two simple arithmetic expressions connected by one of the six relational operators. For instance, if X and Y are numerical variables, then X 'EQUAL' 2*Y is a relation whose value is 'TRUE' whenever X=2Y and 'FALSE' whenever X ≠ 2Y.

TABLE A.4 ALGOL'S LOGICAL OPERATORS (A & B are logical variables)

if A is	and B is	'NOT'A	A'OR'B	A'AND'B	A'IMPL'B	A'EQUIV'B
'TRUE'	'TRUE'	'FALSE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'
'TRUE'	'FALSE'	'FALSE'	'TRUE'	'FALSE'	'FALSE'	'FALSE'
'FALSE'	'TRUE'	'TRUE'	'TRUE'	'FALSE'	'TRUE'	'FALSE'
'FALSE'	'FALSE'	'TRUE'	'FALSE'	'FALSE'	'TRUE'	'TRUE'

Boolean primary: a logical value or variable or anything else whose value works down to 'TRUE' or 'FALSE'. Thus a relation is also a Boolean primary.

Boolean secondary: a Boolean primary or a Boolean primary preceded by 'NOT'.

Boolean factor: two Boolean secondaries (one of which could be a Boolean factor) connected by 'AND'.

Boolean term: Boolean factors connected by 'OR'. One of these factors could be a Boolean term.

Logical operators: the properties of the five logical operators are summarized in Table A.4.

Implication: two Boolean terms connected by 'IMPL'.

Simple Boolean: two implications connected by 'EQUIV'. One of the implications may be a simple Boolean.

Table A.5 shows some samples of Boolean expressions.

A.1.7.6 Designational Expressions

Designational expressions are used to establish statement label values as variable destinations for transfers of control. That is, it is possible to declare an array, where each element is a statement label. Then each label, though it has its own unique name, also has a subscripted name. We shall come back to this later on.

TABLE A.5 BOOLEAN EXPRESSIONS (X is 4, Y is 3, Z is 2, A is 'TRUE', B is 'FALSE')

Expression	Value
X'POWER'2'LESS'4*Y	'FALSE'
SQRT(9*X)'NOTEQUAL'Y'AND'A	'TRUE'
Y'GREATER'X/Z'OR'(B'AND"NOT'A)	'TRUE'
(A'AND"NOT'B'AND"TRUE')'OR'X'EQUAL'Z'POWER'Z	'TRUE'
'TRUE"AND'B	'FALSE'

A.2 BASIC STATEMENTS

The units of operation within ALGOL are called *statements.* Statements may or may not be labeled and may be grouped together into *compound statements* and *blocks.* As is the case with PL/I (Chapter 18), ALGOL source statements are not constrained by any specific coding format restrictions. Columns 1-72 are available for such purposes as long as // or /* do not appear in columns 1-2. Columns 73-80 may be used for statement sequence (ignored by the compiler). Statements end with a semicolon.

A.2.1 Labels

Statement labels are named like identifiers (Sec. A.1.3). The label, followed by a colon, precedes the statement. If the statement is unlabeled, the colon does not appear.

A.2.2 Assignment Statements

The basic form for an ALGOL assignment statement is

v1: = *expression;*

where *v1* is a variable name (identifier) and *expression* is a legitimate ALGOL arithmetic or Boolean expression (Sec. A.1.7). Several variables may appear to the left of *expression.* Thus

```
S:=X:=Y:=2*Z+SQRT(3*S);
```

will cause each of the current values of S, X, and Y to be replaced by $2Z + \sqrt{3S}$.

A.2.3 GOTO Statements

The GOTO statement transfers control to another section of an ALGOL program. Its basic form is

'GOTO' *designational expression;*

Designational expression, defined in Sec. A.1.7.6, defines the destination, in terms of a statement label. When there is no possibility of choice, the statement assumes its basic form, merely

'GOTO' *label;*

A choice of destinations can be provided by using a more extensive desig-

national statement in which a condition is tested and its outcome dictates the label to be used. This takes the form

'IF' *boolean expression* 'THEN' *label*1 'ELSE' *label*2;

For example, the statement

'GOTO''IF'X'LESS'2*LN(4/Y)'THEN'L1'ELSE'L2;

provides the following choice: If X is less than $2ln(4/Y)$, the program will continue executing from the statement labeled L1; otherwise the branch is to L2.

Choices are more extensive when a *switch* is used. This is a list of alternative statement labels. The form for declaring such a switch is

'SWITCH'*name*:=*label*1,*label*2, . . .;

then the GOTO statement could read

'GOTO'*surname* (/*expression*/)

Thus if we said

'SWITCH'ELECT:=L1,L2,L3;
'GOTO'ELECT(/(5*I+1)/8/);

and I was set to a value of 3 at some earlier point, the branch would be to the statement labeled L2. The subscript in this type of statement (in fact, any ALGOL subscript) can be an IF clause. For example, if we wrote

'GOTO'ELECT(/'IF'X'GREATER'3*Y'THEN'I'ELSE'1+2/);

and I had a value of 1, the program would branch to L1 if X exceeds 3Y; the alternative branch, to L3 (the third choice, since I+2 is 3), would be taken if X were not greater than 3Y.

A.2.4 Conditional Statements

The IF clause as used in the previous section may be written as a separate statement:

'IF' *boolean expression* 'THEN' *statement*1 'ELSE' *statement*2

The two alternatives, executed respectively depending on the truth or falsity of *boolean expression*, may be simple statements (like assignment statements) or *compound statements*, where a group of actions will be handled as if they were a single command. A compound statement is set up as follows:

*label*1:'BEGIN' (note that no semicolon is needed after 'BEGIN'.)
statement;

statement;
.
.
statement
'END' *label*1; (since 'END' delimits the group, no semicolon is needed after the preceding statement.)

'BEGIN' and 'END' are known as *dummy statements*. The workings of conditional statements are the same as described for PL/I (Chapter 18).

A.2.5
FOR Statements

Loops in ALGOL are constructed with the FOR statement:

'FOR' *index*:=*n*1,*n*2, . . .'DO'*statement;*

index is a variable whose value will be set to each of the values designated by the list *n*1, *n*2, ... These may be arithmetic expressions of any complexity. Thus, if X is an array, the statement

'FOR'I:=1,3,4'DO'X(/I/):=2*X(/I/)+1;

would be the same as writing

X(/1/):=2*X(/1/)+1;
X(/3/):=2*X(/3/)+3;
X(/4/):=2*X(/4/)+4;

More extensive sequencing is obtained with the form

'FOR' *index*:=*n*1'STEP'*n*2'UNTIL'*n*3'DO'*statement;*

in which case *statement* will be executed for *index* values starting with the value of expression *n*1. Then the value *n*2 (which may also be an expression) is added, and so on until *index* reaches or exceeds *n*3.

Further extension is possible by means of a *while* clause:

'FOR' *index*:=*n*1 STEP *n*2 'UNTIL' *n*3 'WHILE' *boolean expression* 'DO' *statement;*

With this form, the action(s) in *statement* will be performed as long as the condition in *boolean expression* is 'TRUE'. When it is 'FALSE', the *statement* will be skipped, *n*2 will be added to *index* and loop processing continues.

A.2.6
Declarations

Declarations serve to define certain properties or attributes of the quantities used in the program and to associate them with identifiers. A declaration of an identifier is valid for one block; outside this block the

particular identifier may be used for other purposes. A *declaration* is defined as a *type declaration, array declaration, switch declaration,* or *procedure declaration.*

A.2.6.1 Type Declarations

Variables are type 'REAL', 'INTEGER', or 'BOOLEAN'. For example, the statement

```
'INTEGER' A,I,K,N;
```

allocates fixed binary storage for A,I,K, and N.

A.2.6.2 Array Declarations

Arrays with up to 16 dimensions may be declared in ALGOL. Subscripts are specified by giving the lower and upper bounds and may be zero or less. The general form is

type 'ARRAY' *name* (/*n*11:*n*12,*n*21:*n*22 . . . /);

where *n*11:*n*12 are the bounds for the first dimension (rows), *n*21:*n*22 apply to the second dimension, and so on. If *type* is not specified, ALGOL assumes 'REAL'.

Several arrays may be declared in a single statement. Writing

```
'INTEGER''ARRAY'A,B,C(1:5,1:7), D(1:10);
```

defines three 'INTEGER' arrays A,B, and C each with 5 rows and 7 columns, and a ten-element array D.

A.2.6.3 Switch Declarations

In addition to the basic switch declarations shown in Sec. A.2.3, we can arrange more extensive decision mechanisms by allowing several of the choices to depend on some particular set of conditions. For instance, the declaration

```
'SWITCH'WHICH:=L1,L2,'IF'X'EQUAL'3+Y'THEN'L3'ELSE'L4;
```

provides three basic choices: L1,L2 and (L3 or L4), the choice between the latter two depending on the values of X and Y.

A.2.6.4 Procedure Declarations

ALGOL programs are constructed as one or more procedures, each of which has its own name (identifier) and a list of *formal parameters* which correspond to the arguments used when that procedure is called. The

procedure is introduced by the declaration

type 'PROCEDURE' *name* (*parameter, parameter, . . .*);

Specification of the type is optional and procedures may or may not have lists of formal parameters.

EXAMPLE A.1

Write an ALGOL procedure which finds the largest absolute value in a two dimensional M by N array (call it X). The absolute value is to be stored in BIG, its subscripts in ROW and COLM and its sign in HOW.

Solution Program

```
'PROCEDURE'BIGVAL(X,M,N,BIG,ROW,COLM,HOW);
'ARRAY'X;
'INTEGER' M,N,ROW,COLM,HOW;
'REAL'BIG;
'BEGIN''INTEGER'I,J;
BIG:=0;
ROW:=COLM:=1;
'FOR'I:=1'STEP'1'UNTIL'M'DO';
'FOR'J:=1'STEP'1'UNTIL'N'DO';
'IF'ABS(X(/I,J/))'GREATER'BIG'THEN''BEGIN'
                                    BIG:=ABS(X(/I,J/));
                                    ROW:=I;
                                    COLM:=J;
                                    HOW:=SIGN(X(/I,J/))
                                    'END';
'END'BIGVAL;
```

Note that the actual computations are set up in a separate block which includes declarations for I and J, while the other declarations are given outside the 'BEGIN'. This indicates that I and J are *local* to the procedure whereas the other identifiers refer to formal parameters that will be transferred as arguments when the procedure is used. By the same token, X is declared as an array, but no size is given, since no storage is actually reserved for a formal parameter.

The procedure declaration can be extended to include some landmarks for the programmer's convenience. Meaningful names may be assigned to any or all of the formal parameters. The names are ignored in compilation. Thus we could write Example A.1's procedure declaration as

```
'PROCEDURE' BIGVAL(X)ARAYSIZE:(M,N)MAXVALUE:(BIG)
SUBSCRIPTS:(ROW,COLM)SIGN:(HOW);
```

A.2.6.5 Procedure Statements

A procedure is used by means of a procedure statement which refers to the procedure name and passes to it a list of arguments corresponding in type and number to the procedure's formal parameters. The

general form is simply

procname(arguments);

EXAMPLE A.2

Using BIGVAL from Example A.1, write a program that uses two arrays: a 4 × 9 array A1 and a 6 × 8 array A2. The absolute maxima are to be stored in BIG1 and BIG2, the respective subscripts in ROW1 and COL1, ROW2, and COL2, and the signs in HOW1 and HOW2. A value Z is to be calculated as the square root of the sum of the two absolute maxima. (We shall not worry about I/O yet.)

Solution Program

```
'PROCEDURE'MAXADD;
'BEGIN''ARRAY'A1(/1:4,1:9/),A2(/1:6,1:8/);
'INTEGER'ROW1,ROW2,COL1,COL2,HOW1,HOW2,M1,M2,N1,N2;
'REAL'BIG1,BIG2,Z;
M1:=4;M2:=6;N1:=9;N2:=8;
BIGVAL(A1,M1,N1,BIG1,ROW1,COL1,HOW1);
BIGVAL(A2,M2,N2,BIG2,ROW2,COL2,HOW2);
Z:=SQRT(BIG1+BIG2)
END MAXADD;
  'PROCEDURE'BIGVAL(X,M,N,BIG,ROW,COLM,HOW);
  etc;


  'END'BIGVAL;
```

A.3 INPUT/OUTPUT OPERATIONS

The transmission of data to and from an external medium that is not directly accessible by the ALGOL program is achieved by calls to input/output procedures. Data to be transmitted are combined into files or data sets, and each is identified by an integer that may be alternately considered to be an input/output device code. Such integers must be in the range 0–16, where the 0 is reserved for the system's input device SYSIN.

Standard input/output procedures and associated identifiers are built into ALGOL, and these procedures are available without explicit declaration. They are considered as declared in a block surrounding the whole program. There are eighteen standard built-in procedures for input/output processing, and these are described in the following sections.

A.3.1 INSYMBOL and OUTSYMBOL

This pair of procedures treats data transmission in terms of a single symbol consisting of a string of characters. Each procedure requires three arguments: the device code, a string denoting the conversion between the

symbols on the external medium and those in the program, and the variable to or from which transmission is to occur. A one-to-one correspondence is made between the symbols of the string and the positive integers, and these integers replace the actual characters to which they correspond in the symbol being transferred. Where there is no corresponding character on input, a zero is substituted; where a zero appears in a symbol on output, a blank is substituted.

Following is an example showing the conversion of the character sequence BCDAHP68A9ZX4E3FNS to 234180001000050600, the former coming in from device 0 and the latter to appear on device 1:

```
'FOR'I:=1'STEP'1'UNTIL'18'DO';
'BEGIN'INSYMBOL(0,'('ABCDEFGH')',PLACE);
  OUTSYMBOL(1),'('12345678')',PLACE)
'END';
```

If we changed our conversion rules by writing

```
'FOR'I:=1'STEP'1'UNTIL'18'DO';
        'BEGIN'INSYMBOL(0,'('ABCDEFGHPS')',PLACE);
          OUTSYMBOL(1,'('1234567890')',PLACE)
          'END';
```

the same input string would appear on device 1 as 234189␣␣1␣␣␣␣5␣6␣0.

A.3.2
INREAL and OUTREAL

These procedures transfer values of type 'REAL'. For each there are two actual parameters: the device code and the variable (or in the case of OUTREAL, expression) identifier. On input, the first characters encountered in the stream of characters on the file that can be interpreted as numbers are converted to type 'REAL', and the consequent value is assigned to the second actual parameter. On output the value of the specified variable or expression is converted to a field of either 13 or 22 characters, depending on the precision.

As an example, let us consider the action of the program

```
'INTEGER'I;
'REAL'V;
'FOR'I:=1'STEP'1'UNTIL'6'DO';
'BEGIN'INREAL(0,V);
   OUTREAL(1,V)'END';
```

on the input sequence 1, −␣␣034.5␣'5ABC+−7.'7A0␣+'1+X. The six numbers found in this mess would be 1, 34.5×10^{5}, −7, 10^{7}, 0, and 10.

Correspondingly, the six converted output fields would appear with two blanks between adjacent values as follows:

```
+1.000000'+00␣␣+3.450000'+06␣␣-7.000000'+00␣␣
+1.000000'+07␣␣␣0␣␣␣
```

A.3.5
OUTSTRING

This output procedure transfers the symbols between the outermost string quotes of the second actual parameter to the device specified by the first actual parameter. The following two invocations of OUTSTRING produce the character sequence ALGOL REPORT on device 1:

```
OUTSTRING(1,'('ALGOL␣')');
OUTSTRING(1,'('REPORT')');
```

A.3.6
Input/Output of Arrays

For each of the input/output procedures for transmitting data of types 'REAL', 'INTEGER', and 'BOOLEAN', there is a corresponding procedure for transmitting arrays of such variables. Arrays of type 'REAL' are transmitted by INARRAY and OUTARRAY; arrays of type 'INTEGER' are transmitted by INTARRAY and OUTTARRAY; and arrays of type 'BOOLEAN' are transmitted by INBARRAY and OUTBARRAY.

As an example, suppose X is declared as

```
'REAL''ARRAY'X(1:6,1:7);
```

then the procedure statement

```
INARRAY(0,X);
```

would be equivalent to writing

```
'BEGIN''INTEGER' I,J;
  'FOR'I:=1'STEP'1'UNTIL' 6 'DO';
  'FOR'J:=1'STEP'1'UNTIL' 7 'DO';
  INREAL(1,X(/I,J/))
  'END';
```

A.3.7
SYSACT

This procedure is used to obtain finer control over input/output processes. Its general form for usage is

```
SYSACT(n,m,v);
```

where *n* is a device code as before, *m* is an integer from 1–15 representing a *function code* and *v* is a variable or expression whose purpose is dictated by *m*'s value.

Data files in ALGOL are either *open* (available for I/O), *closed* (unavailable for I/O) or *exhausted*, in which case it is available but has no more

TABLE A.6 FUNCTION CODES FOR SYSACT

Function Code Value	v Represents
1	The current value of the character pointer is placed in *v*.
2	*v*'s value is used to set the character pointer.
3	The current value of the record pointer is placed in *v*.
4	*v*'s value is used to set the record pointer; character pointer reset to 1(cannot be used with device 0 or 1.)
5	The file's record length is stored in *v*.
6	*v*'s value is used to set the file's record length (must be used before the file is opened).
7	The file's blocking factor is stored in *v* (0 if unblocked).
8	*v*'s value sets the blocking factor (for an unopened file).
9	The number of blanks placed between adjacent output values (default is 2) is stored in *v*.
10	*v* sets the spacing between adjacent output values for the file.
11	*v* is set to the file's status (1, 0, or −1 for open, closed, or exhausted).
12	A 1 in *v* opens the file; 0 closes it.
13	Same as with a function code of 3, except that the record pointer is stored in an internal index. This index, which can be used later when SYSACT is invoked with a function code of 4, is useful for skipping records.
14	Record pointer is increased by *v*; character pointer set to 1.
15	Skip to next block in the file, record pointer set to *v*'s value, character pointer set to 1.

data on it. A *record pointer* and *character pointer* are maintained for each open file. The former keeps count of the records transmitted to or from a given file and the character pointer indicates the next character (within the current record) to be read or written. When transmission of a record is completed, the record pointer is automatically incremented by 1 and the character pointer is reset to 1. The uses of the function codes are summarized in Table A.6.

EXAMPLE A.3

An array of 'REAL' values consists of 120 values organized as 12 rows and 10 columns. The values are punched one row to a card. Each element is a four-digit integer with the input format being cols. 1-4, 9-12, 17-20, 25-28, 33-36, and so on to 73-76. Replace all elements having odd row numbers and even column numbers with their respective square roots. Print the new array in the same order as read in (all of row 1, then all of row 2, and so forth) five elements to a line. The input device is 0, output is 1. Use four blanks between adjacent output values.

Solution Program

```
'PROCEDURE'CHANGE;
'REAL''ARRAY'X(1:12,1:10);
'INTEGER'I,J,RECIN,RECOUT,INCR,SPACE;START,ADD,CZERO;
```

```
RECIN:=80;RECOUT:=132;INCR:=1;SPACE:=4;START:=25;ADD:=23;
CZERO:=0;
SYSACT(0,6,RECIN);
SYSACT(1,6,RECOUT);
SYSACT(0, ,0);
SYSACT(1,8,0);
SYSACT(0,10,SPACE);
SYSACT(1,10,SPACE);
SYSACT(0,12,INCR); SYSACT(0,2,INCR);
SYSACT(1,12,INCR); SYSACT(1,2,INCR);
INARRAY(0,X);
'BEGIN''FOR'I:=1'STEP'2'UNTIL'11'DO';
'FOR'J:=2'STEP'2'UNTIL'10'DO';

'END';
'BEGIN''FOR'I:=1'STEP'1'UNTIL'12'DO';
  'FOR'J:=1'STEP'5'UNTIL'10'DO';
  SYSACT(1,2,START);
  OUTREAL(1,X(/I,J/));
  OUTREAL(1,X(/I,J+1/));
  OUTREAL(1,X(/I,J+2/));
  OUTREAL(1,X(/I,J+3/));
  OUTREAL(1,X(/I,J+4/));
  SYSACT(1,14,INCR)'END';
'END' CHANGE;
```

A.3.8
GET and PUT

These procedures are used to store (temporarily) and retrieve data on an external medium without any conversion into external formats. The basic form is

```
GET
PUT (n, datalist);
```

where *n* is a device code. *datalisi* is a list of variables to be transmitted.

This will work only when the list of variables is a parameter of a *list procedure.* The procedure is declared with an arbitrary name and its parameter then appears in a 'PROCEDURE' statement.

For example, if X is a 20-element array which is to be stored temporarily on device 12 after having been read from device 0, we could accomplish this as follows:

```
'BEGIN''ARRAY'X(/1:20/);
'PROCEDURE'PUTAWAY(COLLECTION);
'PROCEDURE'COLLECTION;
```

```
  'INTEGER'I;
  'FOR'I:=1'STEP'1'UNTIL'20'DO';COLLECTION (X(/I/));
INARRAY(0,X);
PUT(12,PUTAWAY)'END';
```

SUPPLEMENTARY READING

1. *IBM System/360 Operating System—ALGOL Language,* Form C28-6615. IBM Corp., White Plains, N.Y.
2. *IBM System/360 Operating System—ALGOL Programmer's Guide,* Form C33-4000. IBM Corp., White Plains, N.Y.
3. Schwarz, H. R.; *An Introduction to ALGOL 60, Commun. ACM,* Vol. 5 (February 1962), pp. 82-95.

Appendix B

Introduction To Job Control Language

The various features and options embodied in System/360's Job Control Language (JCL) constitute a major software effort whose complete description is well beyond our present intent. We shall limit this presentation to some of the basic properties.

The job control languages for the various System/360 supervisory programs differ somewhat with respect to their syntax and extent. We shall confine our considerations to the Disk Operating System (DOS) and the full Operating System (OS).

B.1 CODING CONVENTIONS

The Job Control Language has its own relatively unrestricted coding format. Columns 1-2 are always reserved for identifying characters (usually two slashes) which indicate that the information is intended for the Job Control Language analyzer. User-defined job control information must begin in column 3 for OS and in column 4 for DOS, with column 3 being left blank in the latter. Outside of these constraints, the format is quite free, as long as at least one blank is left between fields. In certain cases (as will be indicated herein) it is necessary to separate adjacent fields by a comma.

When a particular job control statement is too long to be accommodated on a single card, additional cards may be used. The first of these may use up to column 71. No field may be broken off on one card and continued on another; instead, any excess columns between the end of a field and column 71 are left blank. If another card is required, any nonblank character is punched in column 72, slashes are punched in cols. 1-2 of the next card, and the statement is continued starting in column 16 and continuing through 71. Columns 3-15 must be blank. Additional cards are handled the same way.

B.2 JOB IDENTIFICATION

The job-identification commands allow the operating system to recognize the beginning of a job, thus allowing it to set up the mechanism for relinquishing control to the appropriate program.

B.2.1 OS Job Identification

All job-identification statements start with // in columns 1-2 and a *job name* starting in column 3. This must begin with a letter and may be up to eight alphameric characters long. The job name is followed by at least one blank and the keyword JOB.

A number of additional parameters may be included. Some of these are listed below:

1. Accounting information, if included, must be the first parameter following JOB. Up to 142 characters may be used for such information. If only an account number is given, it is punched as is. When additional data are provided, the account number, followed by a comma and the other data, are enclosed in parentheses. This may be made compulsory at an installation.
2. The user's name (up to 20 characters), preceded by a comma, may be included. If *any* characters beside letters and periods are used, the name must be enclosed in single quotes.
3. The field MSGLEVEL = 1 will cause all control statements (including those automatically generated by the system) to be displayed. The specification MSGLEVEL = 0 (or no MSGLEVEL specification at all) will allow the display of only the JOB statement on any incorrectly submitted control information.

B.2.2 DOS Job Identification

The DOS job-identification card has // in the first three columns, followed by the keyword JOB starting in column 4 or higher. At least one blank must follow JOB, after which a job name (1-8 characters as in

OS) is specified. The job name may be followed by the keyword LOG (preceded by at least one blank), in which case the job name will be displayed on the console typewriter before and after the job.

B.3 THE EXEC COMMAND

Each job may be composed of one or more *job steps*, wherein each step calls for the execution of a particular program. The EXEC card identifies the beginning of a job step, defines the program to be executed and describes the conditions under which the processing is to occur.

B.3.1 The OS EXEC Card

We shall consider two basic types of programs identified by an EXEC command. If a particular program is part of the system's library or is a private (user-constructed) library it has a unique name associated with it. The EXEC card form for executing such a program is

//stepname ⓑ* EXEC ⓑ PGM = *progname**

whereupon the system searches the library for the program named *progname*.

The system also maintains a *procedure library* (expandable by the user) consisting of *catalogued procedures*. Each of these may be a complex series of programs supported by appropriate JCL commands, including other EXEC statements. Catalogued procedures, like library programs, have unique names consisting of 1–8 characters. When one of these is used, the EXEC card takes the basic form

//stepname ⓑ EXEC ⓑ *catprocname*

Explicit reference may be made to the fact that *catprocname* is a procedure by writing

//stepname ⓑ EXEC ⓑ PROC = *catprocname*

The various assemblers and compilers are included among OS's catalogued procedures. Depending on the information supplied with the EXEC card, the processing may include assembly/compilation, link-editing and execution of the resulting object program. In addition, the user may specify the conditions under which the execution is to proceed. This is done by adding the PARM clause:

//stepname ⓑ EXEC ⓑ *catprocname* ⓑ PARM = *'parm*1, *parm*2, . . .*'*

where *parm*1, *parm*2, . . . represent a list of parameters which define the output produced by a particular compiler, or establish certain format requirements. When a particular parameter is not explicitly defined, a default

*ⓑ represents one or more blanks.

option is used. Table B.1 summarizes these various OS facilities for Assembler Language, FORTRAN, COBOL, PL/I, and ALGOL.

Thus, an EXEC command to compile and execute a PL/I program with no source listing and 60 lines per page would read

```
//step EXEC PL1GO PARM = 'NOSOURCE,LINECNT = 60'
```

The EXEC command must be followed by a card that indicates the device from which the source program is to be read. In most cases this is the system input device (the card reader). Following are the forms for the five languages discussed:

```
//ASM.SYSIN ⓑ DD ⓑ *
//FORT.SYSIN ⓑ DD ⓑ *
//COB.SYSIN ⓑ DD ⓑ *
//PL1.SYSIN ⓑ DD ⓑ *
//ALGL.SYSIN ⓑ DD ⓑ *
```

This is followed by the source deck, whose last statement is followed by a card with /* in columns 1–2 to signify the end of the source program.

When a user's previously assembled object program is to be executed under OS, and that program is not part of a library, the sequence is as follows:

```
//jobname ⓑ JOB ⓑ etc
//stepname ⓑ EXEC ⓑ LINKGO
//LKED.SYSLZN ⓑ DD ⓑ *
  object deck
/*
```

B.3.2 DOS EXEC Specifications

The conditions under which DOS programs are to be executed are specified as part of the OPTION instruction, which precedes the EXEC card. Its general form is

```
// ⓑ OPTION ⓑ option, option, . . . .
```

Table B.2 lists the features that can be specified.

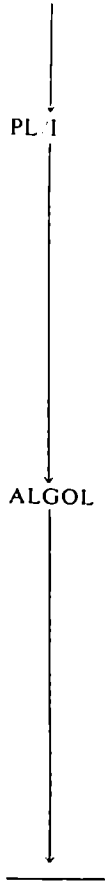
With this information having been relegated to the OPTION instruction, the EXEC card itself becomes rather simple:

```
// ⓑ EXEC ⓑ ASSEMBLY
// ⓑ EXEC ⓑ FORTRAN
// ⓑ EXEC ⓑ COBOL
// ⓑ EXEC ⓑ PL1
no ALGOL under DOS
```

*Additional JCL cards, as discussed later.

TABLE B.1 PROCEDURE NAMES AND PARAMETERS FOR OS EXEC COMMANDS

Language	Catalogued Procedure Name	PARM Name	Action
Assembler	ASSEM	none specified	Source deck is assembled, object deck produced.
		DECK* or NODECK	Controls production of physical object deck.
		LOAD or NOLOAD*	Controls preparation of object program for link-editor.
		LIST* or NOLIST	Controls printout of assembly listing.
		XREF* or NOXREF	Controls printout of cross-reference listing.
		LINECNT = *nn*	Controls number of lines per page; default is 55.
	ASSEMGO		Specifies assembly, link-edit and execute. Parameters operate as with ASSEM.
FORTRAN (levels E,G or H)	FORTE,FORTG or FORTH	None specified	Source deck compiled, object deck not produced.
		DECK or NODECK*	Same as for assembler (note different default).
		LOAD* or NOLOAD	Same as for assembler (note different default).
		SOURCE* or NOSOURCE	Controls printout of source listing.
		MAP or NOMAP*	Controls printout of object program's memory map.
		SIZE = *nnnnnnn*	Amount of core available to compiler; default set by installation.
		LINELNG = *nnn*	Number of print positions per line; default is usually 132.
	FORTEGO,FORTFGO or FORTHGO	NAME = *aaaaaa*	Program name; default is MAIN, specifies compilation, link-edit and execution. Parameters as with FORTE, FORTG, FORTH.
COBOL	COBOL (F Level)	None specified	Source deck compiled, object deck produced.
		DECK* or NODECK	Same as for assembler.
		LIST* or NOLIST	Same as for assembler.
		DMAP* or NODMAP	Controls printout of list of data name addresses.
		PMAP* or NOPMAP	Controls printout of object code listing.
		MAPS* or NOMAPS	Combined effect of DMAP and PMAP.
		DISPCK* or NODISPCK	Object program will test field lengths in DISPLAY.



	LINECNT = *nn*	Same as for Assembler.
	BUFSIZE = *nn*	Size of buffers for compiler; default is set by installation.
COBOLGO		Specifies compilation, link edit and execution. Parameters as with COBOL.
PL/I	None specified	Source deck is compiled, object deck not produced.
PL1	DECK or NODECK*	Same as for Assembler.
	LOAD* or NOLOAD	Same as for Assembler.
	LIST* or NOLIST	Controls listing of object program.
	SOURCE* or NOSOURCE	Same as for FORTRAN.
	EXTREF or NOEXTREF*	Controls listing of external symbol dictionary.
	ATR* or NOATR	Controls listing of identifiers' attributes.
	XREF or NOXREF*	Controls listing of cross reference table.
	CHAR60* or CHAR48	Specifies character set used.
	SIZE = *nnnnn*	Core size available to compiler; default set by installation.
	LINECNT = *nn*	Number of lines per page; default usually set at 55.
	SORMGIN = (*n,m*)	Margins for source statements; default usually set at 2 and 80.
PL1GO	—	Compilation, link edit, execution. Parameters are as in PL1.
ALGOL ALGOFC (level F)	None specified	Compilation only.
	DECK or NODECK*	Same as for Assembler.
	LOAD* or NOLOAD	Same as for Assembler.
	SOURCE* or NOSOURCE	Same as for FORTRAN.
	LIST or NOLIST*	Controls listing of job control statement.
	MAP or NOMAP*	Same as in FORTRAN.
	XREF or NOXREF*	Same as in Assembler.
	SHORT* or LONG	Controls the precision of 'REAL' values.
	PROGRAM* or PROCEDURE	Specifies whether step is a complete program or separately compiled procedure.
	SIZE = *nnnnn*	Same as in COBOL.
ALGOFCL	—	Compilation and link-edit.
ALGOFCLG	—	Compilation, link-edit, and execution.

* Default option

TABLE B.2 PARAMETERS FOR THE DOS OPTION INSTRUCTION

Language	Parameter	Action
General	LOG	Control statements are printed.
	DUMP	Contents of memory are displayed if program ends before it should.
	LINK	Object program is prepared for link editing (NODECK assumed).
	DECK	Object deck is produced (NOLINK is assumed).
	LIST	Source program is listed.
	SYM	Symbol table is printed.
	XREF	Cross-reference list is printed.
	CATAL	Object program is catalogued.
COBOL	ERRS	All source errors are listed.
	SYM	Data division map is printed.
	LISTX	Object program is listed.
PL/I	48C or 60C*	Specifies the character set used.

*Default option.

If we wish to assemble/compile, link-edit, and execute under DOS in automatic sequence, we must set up for three executions (the assembler or compiler, the link editor and, finally, the user's object program itself. The basic form is as follows (using PL1):

```
// ⓑ JOB ⓑ jobname
// ⓑ OPTION ⓑ LINK
// ⓑ EXEC ⓑ PL1
  PL/1 source program
/*
// ⓑ EXEC ⓑ LINKEDT
// ⓑ EXEC
additional cards as described later
```

If an object deck is already available, then we can submit:

```
// ⓑ JOB ⓑ jobname
// ⓑ OPTION ⓑ LINK
// ⓑ EXEC ⓑ LINKEDT
  object deck
/*
// ⓑ EXEC
```

Note that in either case the link editor positions a module for execution, so that the plain EXEC card is sufficient, since it causes the positioned program to execute.

B.4 DEVICE ASSIGNMENT

Certain physical devices are given particular names in the supervisory system, so that it is possible to use an I/O statement without specifying the external source or destination of the data. When this happens, the system assigns a standard device. For example, the PL/I statement

```
GET LIST (data list);
```

is really an implicit way of saying

```
GET FILE (SYSIN) LIST (data list);
```

When additional data files are to be read from or written on devices other than those defined as *standard* by the system, additional JCL information must be included. This information names the collection of data, associates it with a particular physical device and describes the organization of the data.

B.4.1 The DD Statement for OS

The DD (Data Device) statement is the most extensive JCL command. By using proper combinations of options, it may be used to create new files, recognize existing ones and add to or delete from the catalog of *data sets*. A data set in this context is a particular collection of data associated with a specific device. Under OS it is possible to include such data sets as part of the same general library that contains programs and procedures.

Regardless of the higher-level language being used, an I/O statement uses a particular name to define the external source or destination of data. For example, when a COBOL statement says

```
READ MASTFL RECORD INTO PLACE.
```

the collection has been given the name MASTFL. That name is known as a dd*name* and is linked through the DD statement with a physical device and a specific description.

The basic form for the DD statement is as follows:

```
//GO.ddname ⓑ DD ⓑ options
```

Some of these options and their values are summarized in Table B.3.

As an example, let us set up a DD instruction for an existing tape file named TREF that resides on a reel labeled MSTR. The tape is written in even parity BCD and is to be translated to EBCDIC. Label procedures are to be by-passed. The records are fixed-length, consisting of 231 characters each, blocked 5. The tape is to be mounted on Unit 281.

TABLE B.3 DD OPERAND OPTIONS IN OS

Operand	Purpose
DSNAME = *name*	Defines the name assigned to a particular data set. For example, if we're talking about a tape to be used by a PL/I program, then //GO. PYMSTR DD DSNAME = MSTR3 says that the file named PYMSTR is on a reel of tape whose physical label (the one pasted on, not written on the tape itself) is MSTR3.
DISP = *status, dispostion*	Describes how the file is to be handled. *Status* NEW (this is to be newly created) OLD (the data collection already exists) MOD (the data collection will be modified)
	Disposition KEEP (retain after the job is done) DELETE (destroy after the job is done) PASS (make available to another job step)
UNIT = *unit data*	The unit may be described by an address assigned to a specific device at an installation (UNIT = 180) or it may be a particular type (UNIT = 1403).
VOLUME = *volume data*	This refers to the particular volume (a tape reel) that houses the data file. The most common form is VOLUME = REF = *dsname*
LABEL = *label data*	NL (no labels) SUL (both standard and user labels) SL (standard labels) NSL (nonstandard labels) BLP (bypass label processing)
	Describes data organization; general form is (*parameter* = *value*, *parameter* = *value*, . . .).

DCB = *data control block information*

Parameter	Purpose
BLKSIZE = *nn*	Maximum Block Size in Bytes.
CYLOFL = *nn*	For disks, number of tracks to be reserved on each cylinder to handle overflow records from other tracks on that cylinder.
DEN = 0,1 or 2	Tape Density (200, 556, or 800 bpi).
DSORG = *code*	PS (physical sequential) IS (indexed sequential) DA (direct access) } data-set organization
KEYLEN = *nn*	Number of bytes used for a key.
LRECL = *nn*	Maximum number of bytes in a logical record.
RECFM = *code*	U (undefined) V (variable record length) F (fixed record length) B (blocked) T (track overflow used) } Record format
TRTCH = *code*	Tape-recording technique C (data-conversion feature) E (even parity) T (BCD to EBCDIC translation)

Our DD statement would read

```
1                16                                                    72
//GO.TREF DD  DSNAME=MSTR,DISP=(OLD,KEEP),                           *
//           LABEL=(,BLP),VOLUME=SER=MSTR,                            *
//           UNIT=281, DCB=(RECFM=FB,BLKSIZE=1155,
//           LRECL=231,DEN=1,TRTCH=ET)
```

In FORTRAN, the *dd* name takes the form FT*nn*F001 where *nn* is a number from 1 to 99. If we assume that *nn* is 12 for our example, then the beginning of our DD statement would read

```
//GO.FT12F001 DD DSNAME=MSTR,etc.
```

and, if the file is an input file, it would be read with

```
READ(12,m)list
```

A complete JCL description is given in Ref. 1. These DD cards (a set for each data set) follow the /* card that signals the end of the source or object deck (if one is submitted). The data, if any are submitted on cards, follow the DD cards; irrespective of the presence or absence of card data, the job ends with another /* card. Thus a PL/I compiler, link-edit and execute job in which data will be coming from cards as well as from this file TREF would be submitted as follows:

```
//jobname JOB ....
//step EXEC PL1GO
//PL1.SYSIN DD *
  source deck
/*
//GO.TREF DD DSNAME=MSTR,etc.
//.GO.SYSIN DD *
data cards
/*
```

B.4.2 The ASSGN Statement in DOS

As is the case with OS, DOS makes certain standard assignments. The various functional devices have coded names beginning with SYS, as summarized in Table B.4. In addition, the names SYS000-SYS244 may be associated with specific devices. This is usually done when the system is activated (at the beginning of the day) by means of ASSGN statements. The general form is

```
// ⓑ ASSGN ⓑ SYSnnn, X'device m'
```

TABLE B.4 DOS DEVICE NAMES

Name	Function
SYSRES	Systems residence unit (usually a 2311 disk unit).
SYSLOG	Device for displaying messages to the operator (usually the console typewriter).
SYSRDR	Systems reader, which is the device from which control cards are read (usually a card reader).
SYSIPT	Systems input unit (may be the same as SYSRDR).
SYSLST	Device for displaying output (usually the printer).
SYSPCH	Systems punch (may be same device as SYSLST).
SYSLNK	Input device to the linkage editor (usually an area of a 2311 disk).

Thus if an installation is set up such that a particular tape unit is permanently numbered 180, the statement

```
// ASSGN SYS004,X'180'
```

will assign the name SYS004 to device 180. The ASSGN cards (one for each SYS*nnn*) are placed between the JOB and EXEC cards.

If it turns out that a user wishes to change the correspondence between a SYS*nnn* value and a particular device, he does so the same way. The new assignments will hold until a new ASSGN is met. The addition of the keyword TEMP, preceded by a comma, will limit the device assignments to that particular job.

In DOS FORTRAN, there is a set relation between SYS*nnn* and FORTRAN device number: Device no. = SYSno.-3. Thus the statement

```
READ (8,m) list
```

will read from SYS005. For example, if we wanted to compile, link-edit, and execute a FORTRAN program in which device 280 is to be assigned to SYS006, the job would be set up as follows:

```
// JOB jobname
// OPTION LINK
// EXEC FORTRAN
  source program
/*
// EXEC LINKEDT
//ASSIGN SYS006,X'280'
//EXEC
data
/*
/&      (end-of-job card)
```

and we would read with READ (3,*m*) *list*.

ADDITIONAL READING

1. *IBM System/360 Disk Operating System—Concepts and Facilities* —Form C24-5030. IBM Corp., White Plains, N.Y.
2. *IBM System/360 Disk Operating System—Data Management Concepts*-Form C24-5037. IBM Corp., White Plains, N.Y.
3. *IBM System/360 Disk Operating System—System Control and Service Programs*-Form C24-5036. IBM Corp., White Plains, N.Y.

REFERENCES

1. *IBM System/360 Operating System—Job Control Language*—Form C28-6539. IBM Corp., White Plains, N.Y.

Index

Access time, 19
Address modification, 160
Addressing methods, 49
ALGOL, arguments in, 536
  array declarations in, 541
  arrays in, 535
  assignment statements, 538
  basic symbols in, 532
  BEGIN statement, 539
  Boolean expressions, 536
  built-in functions in, 536
  character set for, 532
  character strings in, 534
  coding rules, 532
  comments in, 533
  cyclic operations in, 540
  declarations in, 540
  declarators, 533
  designational expressions, 537
  DO loops in, 540
  END statement, 540
  expressions in, 535
  FOR statement, 540
  GET statement, 548
  GOTO statement, 538
  identifiers, 533
  IF statement, 538
  INBOOLEAN specification (for input/output), 545
  ININTEGER specification (for input/output), 545
input/output function codes for, 547
input/output of arrays, 546
ALGOL (*cont.*)
  input/output operations in, 543
  INREAL specification (for input/output), 544
  INSYMBOL specification (for input/output), 543
  INTEGER declaration, 534
  operators in, Table A.1, 533
  organization of programs in, 541
  OUTBOOLEAN specification (for input/output), 545
  OUTINTEGER specification (for input/output), 545
  OUTREAL specification (for input/output), 544
  OUTSYMBOL specification (for input/output), 543
  PROCEDURE statement, 542
  PUT statement, 548
  REAL declaration, 534
  representation of numbers in, 534
  sequential operators in, 533
  specificators, 533
  statement labels, 538
  subscripts in, 535
  switches in, 539
  use of pointers, in input/output operations, 547
  variable names in, 533
  WHILE keyword (with FOR), 540
Arguments, and built-in functions, 463
  in ALGOL functions, 536
  in FORTRAN built-in functions, 295

Arguments (*cont.*)
in PL/I built-in functions, 463
in PL/I subprograms, 515
Arithmetic operations, built-in functions for, in PL/I, 464
in ALGOL, 536
in Assembler Language, 129, 172, 209
in COBOL, 366, 384
in FORTRAN, 293
in PL/I, 462
on arrays, in PL/I, 466
on complex numbers, 346
on decimal numbers, 170
on floating-point numbers, 209
Arrays, built-in functions for, in PL/I, 469
cross sections of, in PL/I, 467
in ALGOL, 535, 541
in COBOL, 409
in FORTRAN, 298
input/output of, in ALGOL, 546
input/output of, in FORTRAN, 318
input/output of, in PL/I, 497
manipulation of, in FORTRAN, 307
manipulation of, in PL/I, 466, 469
organization of, in PL/I, 466
storage allocation for, in Assembler Language, 122, 171
storage allocation for, in FORTRAN, 300
storage allocation for, in PL/I, 465
subscripts for, in ALGOL, 535
subscripts for, in COBOL, 410
subscripts for, in FORTRAN, 299
subscripts for, in PL/I, 466, 468
ASCII, data representation in, 47
ASCII mode indicator, 85
Assembler, basic functions of, 94
Assembler instructions, 102
Assembler Language, A instruction, 129
absolute terms in, 108
accessibility of location counter in, 107
AD instruction, 209
address modification in, 160
ADR instruction, 209
Assembler Language (*cont.*)
AE instruction, 209
AER instruction, 209
AH instruction, 131
AL instruction, 236
ALR instruction, 236
AP instruction, 172
AR instruction, 134
AU instruction, 209
AUR instruction, 209
AW instruction, 209
AWR instruction, 209
BALR instruction, 127
basic organization of programs in, 127
BC instruction, 148
BCR instruction, 150
BCT instruction, 162
BCTR instruction, 163
boundary alignment in, 117
BXH instruction, 167
BXLE instruction, 165
C instruction, 154
CCB macro, 261
CCW instruction, 124, 256
CD instruction, 210
CDR instruction, 210
CE instruction, 210
CER instruction, 210
CH instruction, 154
character manipulation in, 183
character set for, 95
CL instruction, 243
CLC instruction, 243
CLI instruction, 243
CLOSE macro, 282
CLR instruction, 243
CNOP instruction, 269
coding conventions for, 110
comments, 109
COPY instruction, 274
CP instruction, 193
CR instruction, 154
CSECT instruction, 271
D instruction, 142
DC instruction, 112, 171
DD instruction, 210
DDR instruction, 210

Assembler Language (*cont.*)
DE instruction, 210
definition of constants in, 112
DER instruction, 210
DP instruction, 180
DR instruction, 142
DROP instruction, 271
DS instruction, 122, 171
DSECT instruction, 272
DTFDA macro, 279
DTFSR macro, 275
ED instruction, 200
EDMK instruction, 203
EJECT instruction, 269
END instruction, 128
ENTRY instruction, 271
EOJ instruction, 128
EQU instruction, 123, 268
EX instruction 243
EXCP macro, 261
EXTRN instruction, 271
floating-point operations in, 207
GET macro, 284
HDR instruction, 210
HER instruction, 210
HIO instruction, 257
IC instruction, 234
ICTL instruction, 268
indexing in, 164
ISEQ instruction, 273
ISK instruction, 259
L instruction, 128
LA instruction, 237
labeling of statements in, 97
LCR instruction, 136
LD instruction, 209
LDR instruction, 209
LE instruction, 209
length attribute in, 107
LER instruction, 209
LH instruction, 131
literals in, 105
LM instruction, 134
LNR instruction, 135
logical operations in, 226
looping techniques in, 161
LPDR instruction, 213
LPER instruction, 213

Assembler Language (*cont.*)
LPR instruction, 135
LPSW instruction, 260
LR instruction, 132
LTDR instruction, 216
LTER instruction, 216
LTORG instruction, 274
LTR instruction, 148
M instruction, 137
macro instructions in, 260
manipulation of binary values in, 126
masking operations in, 152
MD instruction, 215
MDR instruction, 215
ME instruction, 215
MER instruction, 215
MH instruction, 137
MP instruction, 180
MR instruction, 137
MVC instruction, 183
MVI instruction, 184
MVN instruction, 185
MVO instruction, 186
MVZ instruction, 186
N instruction 229
NC instruction, 229
NI instruction, 228
NR instruction, 229
numerical constants in, 104
O instruction, 231
OO instruction, 231
OI instruction, 231
OPEN macro, 281
operand expressions in, 107
OR instruction, 231
ORG instruction, 268
organization of mnemonic codes in, 98
organization of programs in, 271
PACK instruction, 174
packed-decimal operations in, 170
PRINT instruction, 273
PRTOV macro, 286
pseudo-instructions in, 102
PUNCH instruction, 273
PUT macro, 284
PUTX macro, 286

Assembler Language (*cont.*)
random number generation in. 237
RDD instruction. 258
READ macro. 283
relocatable terms in. 108
RELSE macro. 286
REPRO instruction. 274
rounding in. 189
S instruction. 132
SD instruction. 213
SDR instruction. 213
SE instruction. 213
self-defining terms in. 104
sequencing of statements in. 110
SER instruction. 213
SH instruction. 145
SIO instruction. 255
SL instruction. 236
SLA instruction. 145
SLDA instruction. 140
SLL instruction. 240
SLR instruction. 236
SP instruction. 173
SPACE instruction. 269
SPM instruction. 259
SRA instruction. 145
SRC instruction. 134
SRDA instruction. 140
SRDL instruction. 240
SRL instruction. 240
SSK instruction. 259
SSM instruction. 259
ST instruction. 129
START instruction. 127. 268
STC instruction. 234
STD instruction. 209
STE instruction. 209
STH instruction. 131
STM instruction. 132
storage allocation in. 112
structure of statements in. 96
SU instruction. 213
subroutines in. 155
SUR instruction. 213
SVC instruction. 260
SW instruction. 213
SWR instruction. 213
symbols in. 99

Assembler Language (*cont.*)
table of instructions in. 100
TCH instruction. 258
TIO instruction. 258
TITLE instruction. 272
TM instruction. 151
TR instruction. 199
TRT instruction. 196
TRUNC macro. 286
TS instruction. 259
UNPK instruction. 175
USING instruction. 128. 270
WAIT macro. 262
WDD instruction. 259
WRITE macro. 283
X instruction. 232
XC instruction. 232
XI instruction. 232
XR instruction. 232
Assignment statements. in ALGOL. 538
in FORTRAN. 293
in PL/I. 462
Automatic job handling. 4

Base-displacement addressing. 51
Binary numbers. in ALGOL. 534
in Assembler Language. 207
in FORTRAN. 292
in PL/I. 458
representation of. 56
Bit strings. in PL/I. 473. 477
Boolean operations (*See* Logical operations)
Boundary alignment. 117
Branching. in ALGOL. 538
in Assembler Language. 147. 156
in COBOL. 370
in FORTRAN. 300
in PL/I. 476
role of condition code in. 148
Buffering. in PL/I. 504
Built-in functions. arguments for. 463
for arrays. in PL/I. 469
for bit handling. in PL/I. 475
for character handling. in PL/I. 472
for complex operations. (FORTRAN). 348

Built-in functions (*cont.*)
in ALGOL, 536
in FORTRAN, 295
in PL/I, 463

Card readers and punches, operating characteristics of, 21
Cathode Ray Tube (CRT) devices, operating characteristics of, 21
Central Processing Unit (*See* CPU)
Channel Address Word (CAW), 89, 248
Channel Command Word (CCW), 90, 249
Channel Commands, 249
Channel Status Word (CSW), 90, 251
Channel-to-channel adapter, 13
Character manipulation, in Assembler Language, 183
in COBOL, 406
in PL/I, 471
Character set, for ALGOL, 532
for Assembler Language, 95
for COBOL, 360
for FORTRAN, 288
for PL/I, 455
Character strings, in ALGOL, 534
in Assembler Language, 112, 171
in PL/I, 459, 471
COBOL, ACCEPT verb, 369
ACCESS specification, 376, 400
ACTUAL KEY specification, 401
ADD verb, 366
ALL specification, 382
ALTER verb, 370
ANY specification, 382
APPLY RANDOM PROCESSING specification, 445
APPLY RESTRICTED SEARCH specification, 415
arithmetic operations in, 366
array specifications in, 409
ASCENDING SPECIFICATION (for sorting), 428
ASSIGN verb, 375
AUTHOR specification, 361
BASIS specification (with source program library), 450

COBOL (*cont.*)
BLOCK CONTAINS specification, 377
branching in, 370
BY keyword (with PERFORM), 411
CALL verb, 418
character manipulation in, 406
character set for, 360
CHARACTERS keyword, 405
CLOSE verb, 379
coding conventions for, 359
COLUMN specification (for report writing), 436
comments in, 361
comparison operations in, 371
COMPUTATIONAL specification, 362, 364
COMPUTE verb, 366
CONFIGURATION SECTION, 361
CONTROL specification (for report writing), 433
COPY feature (with source program library), 449
DATA DIVISION, 362, 395
DATA RECORD specification, 377
DATE-COMPILED specification, 361
DATE-WRITTEN specification, 361
debugging features, 448
DECLARATIVES specification, 421
DELETE specification (with source program library), 450
DEPENDING ON specification, 370, 410
DESCENDING specification (for sorting), 428
DETAIL specification (for report writing), 434
DIRECT-ACCESS specification, 375
direct files, 400
disk file characteristics in, 400
DISPLAY verb, 367
DIVIDE verb, 385
editing of, output in, 398, 432
ENTER COBOL specification, 418
ENTER LINKAGE specification, 417
ENTRY specification, 417
ENVIRONMENT DIVISION, 361
error-handling techniques in, 420
EXAMINE verb, 382

COBOL (*cont.*)
EXIT verb, 423
expanded IF statement, 448
figurative constants, 381
FILE CONTROL specification, 375, 361
file handling in, 399, 422
file searching in, 415
FILE-UNIT specification, 447
files on magnetic tape, 379
FIRST DETAIL specification (for report writing), 433
floating-point numbers in, 396
FOOTING specification (for report writing), 433
FROM CONSOLE option (with ACCEPT), 369
FROM keyword (with PERFORM), 411
FROM keyword (with SUBTRACT), 367
FROM keyword (with TRANSFORM), 405
GENERATE verb (for report writing), 437
GO TO statement, 367, 370
HEADING specification (for report writing), 433
IDENTIFICATION DIVISION, 361
IF statement, 371
INCLUDE specification (with source program library), 450
indexed files, 400
initialization of values in, 365
INITIATE verb (for report writing), 437
Input-Output Section, 361, 376
INPUT PROCEDURE, 429
INSERT specification (with source program library), 450
INSTALLATION specification, 361
INVALID KEY specification, 404
I/O CONTROL specification, 361
JUSTIFIED RIGHT specification, 423
LABEL RECORDS specification, 377
LAST DETAIL specification (for report writing), 433
LEADING specification, 382
LINES specification (for report writing), 433

COBOL (*cont.*)
LINKAGE SECTION, 418
literal values in, 362
loops in, 410
MOVE verb, 369
MULTIPLY verb, 384
NEXT GROUP specification (for report writing), 434
OBJECT-COMPUTER specification, 361
OCCURS specification, 409
ON SIZE error, 367
OPEN verb, 379
ORGANIZATION specification, 400
output format control in, 385
OVERFLOW specification (for report writing), 434
PAGE specification (for report writing), 433
PERFORM verb, 410
PICTURE specification, 364, 395
PROCEDURE DIVISION, 362
PROCEED verb, 370
program organization, 361
PROGRAM-ID specification, 361
RANDOM specification, 376, 400
RECORD CONTAINS specification, 377
RECORDING MODE specification, 376
RECORD KEY specification, 400
REDEFINE specification, 423
relative files, 400
RELEASE specification (for sorting), 428
REMARKS, 361
REPLACING specification, 383
REPORT SECTION, 432
report writing features, 431
RERUN verb, 422
RESERVE specification, 375
RESET ON FINAL specification (for report writing), 436
RESET specification (for report writing), 436
RETURN verb, 418
REWRITE verb, 420
ROUNDED option, 366
SAME specification, 422
SECURITY specification, 361
SELECT specification, 400

COBOL (*cont.*)
SELECT verb, 375
sequential files, 400
SEQUENTIAL specification, 376, 400
SORT feature, 428
sorting in, 426
SOURCE-COMPUTER specification, 361
source program library, 449
SOURCE specification (for report writing), 436
SPECIAL-NAMES specification, 446
storage allocation in, 364
subprograms in, 417
subscripts in, 410
SUBTRACT verb, 367
SUM specification (for report writing), 436
SYMBOLIC KEY specification, 401
TALLYING option, 382
TERMINATE verb (for report writing), 437
THRU keyword (with PERFORM), 410
TO keyword, 366, 369
TO keyword (with TRANSFORM), 405
TRACK-AREA specification, 447
TRANSFORM verb, 405
TYPE specification (for report writing), 434
UNIT-RECORD specification, 375
UNTIL FIRST specification, 383
UNTIL specification (with PERFORM), 411
UPON CONSOLE option (with DISPLAY), 367
UPON SYSPUNCH option (with DISPLAY), 367
USAGE keyword, 365
USE FOR RANDOM PROCESSING specification, 445
USE specification (with declaratives), 421
USING specification (with ENTRY), 417
UTILITY SPECIFICATION, 375
VALUE IS specification, 365
VALUE specification, 365
variable-length records in, 423
variable names in, 423
COBOL (*cont.*)
VARYING specification (with PERFORM), 411
WITH CODE specification, 446
WORKING-STORAGE SECTION, 362
WRITE verb, 385
Coding rules, for ALGOL, 532
for Assembler Language programs, 110
for COBOL, 359
for FORTRAN, 288
for JCL, 550
for PL/I, 454
Comments, in ALGOL programs, 533
in Assembler Language programs, 109
in COBOL programs, 361
in PL/I programs, 457
Complex numbers, in FORTRAN, 346
in PL/I, 458
Complements, 58
Concatenation, in PL/I, 471
Condition code, 86, 148
Constants, definition of, in Assembler Language, 112
figurative, in COBOL, 381
in ALGOL, 532
in FORTRAN, 291
in PL/I, 457
CPU, interfaced with I/O channels, 9
organization of, 15
size of, 16
speed of, 19
Cyclic operations, in ALGOL, 540
in Assembler Language, 161

Data, hierarchical organization, in PL/I, 469
hierarchical structures, in COBOL, 378
organization for direct-access devices, 280
organization for magnetic tape, 279
organized as COBOL records, 376
organized as records, 282
organized as records, in PL/I, 504
Data cells, 27

Data compression, 241
Data Control Block, in JCL, 558
Data set, organization of, 34
Data-directed input/output, in PL/I, 489
Decimal numbers, internal representation of, 170
Declarators, in ALGOL, 533
Direct-access devices, operating characteristics of, 27, 30
  organization of data for, 280
Disk drives, 27
Doubleword, 49
Drums, 27
Dynamic storage allocation, in PL/I, 524

EBCDIC, data representation in, 45
Edit-directed input/output, in PL/I, 490
Editing, in ALGOL, 547
  in Assembler Language, 199, 203
  in COBOL, 398, 432
  in FORTRAN, 314
  in PL/I, 488, 493
Extended alphabet, 95
External interrupts, 82

Fetch protection, 20
Files, attributes of, in COBOL, 379
  buffered and unbuffered, in PL/I, 504
  declaration of, in PL/I, 499
  definition of, in Assembler Language, 275
  description of, in COBOL, 400
  manipulation of, in COBOL, 399, 422
  names for, in PL/I, 502
  opening and closing, 281
  organization for direct-access devices, 280
  organization for magnetic tape, 279
  organization of, in COBOL, 376
  organization of, under JCL, 558
  searching techniques for, in COBOL, 415
Files (*cont.*)
  sorting techniques for, in COBOL, 428
  specification of, in COBOL, 376
Floating-point numbers, in ALGOL, 534
  in Assembler Language, 207
  in COBOL, 396
  in FORTRAN, 292
  in PL/I, 458, 460
  internal representation of, 60
FORTRAN, arithmetic expressions in, 293
  arithmetic IF statement, 301
  arrays in, 298
  ASSIGN statement, 355
  assigned GO TO statement, 355
  assignment statements, 293
  BACKSPACE statement, 327
  blank vs. labeled common, 355
  BLOCK DATA subprogram, 355
  built-in functions, 295
  CALL statement, 339
  coding conventions for, 288
  comments in, 289
  COMMON statement, 343
  complex numbers in, 346
  COMPLEX specification, 346
  computed GO TO statement, 301, 355
  CONTINUE statement, 306
  cyclic operations in, 304
  declaration of variables, 292
  DIMENSION statement, 300
  display of characters, 321
  DO loops, 304, 318
  DOUBLE PRECISION specification, 292
  double precision variables, 291
  DUMP subprogram, 340
  DVCHK subprogram, 340
  editing output in, 314
  END FILE statement, 327
  EQUIVALENCE statement, 344
  EXIT subprogram, 340
  expanded READ statement, 353
  EXTERNAL statement, 344
  FORMAT statement, 314

FORTRAN (*cont.*)
FUNCTION statement, 335
GO TO statement, 300
IF statement, 301
IMPLICIT specification, 347
INTEGER specification, 292
integer variables, 291
keywords in, 291
literal values in, 322
logical declaration, 347
logical expressions, 351
logical IF statement, 351
logical operations, 349
matrix manipulation, 307
NAMELIST statement, 354
nested DO loops, 307
organization of programs in, 345
OVERFL subprogram, 340
PAUSE statement, 302
PDUMP subprogram, 341
READ statement, 317
REAL specification, 292
real variables, 291
relational operators, 349
representation of numbers in, 291
RETURN statement, 335
REWIND statement, 327
scaling of output, 325
SLITE subprogram, 339
SLITET subprogram, 340
statement function subprograms, 332
statement labels for, 289
STOP statement, 302
storage allocation in, 292
subprograms in, 332
SUBROUTINE statement, 338
subscripting rules in, 299
tape operations in, 327
variable names in, 291, 347
WRITE statement, 314
Fullword, 49
Function procedures, in PL/I, 514

Generic functions, in PL/I, 464

Halfword, 49
Hardware, basic organization of, 11
Hexadecimal numbers, 52
Hypertape drive, 26

Indexing, 51
Input/Output (*See* I/O)
Instruction address, 87
Instruction formats, RR format, 65
RS format, 67
RX format, 66
SI format, 68
SS format, 69
Instruction length code, 86
Interrupt systems, 7
Interruption codes, 86
Interrupts, classes of, 81
enabling and disabling of, in PL/I, 522
handling of, in PL/I, 503, 522
masking of, 82
preparation for handling, 259
role of PSW in, 87
I/O channels, and the Channel Address Word, 89
and the Channel Command Word, 90
and the Channel Status Word, 90
and overlapped operation, 5
communication between, 13
interfaced with the CPU, 9
multiplexer channel, 11, 15
selector channel, 11, 14
I/O equipment, card readers and punches, 21
Cathode Ray Tube (CRT) devices, 21
data cells, 29
magnetic disk drives, 27
magnetic drums, 27
magnetic tape devices, 24
printers, 21
I/O interrupts, 81
I/O operations, Channel Address Word in, 248
Channel Command Word in, 249
Channel Commands for, 249
Channel Status Word, 251
channel testing (TCH), 258
defining files for, 275

I/O operations (*cont.*)
function codes for, in ALGOL, 547
in ALGOL, 543
in COBOL, 375, 380, 385
in FORTRAN, 313
in PL/I, 486
initiation of (SIO), 255
Read Direct (RDD), 258
records, 282
termination of (HIO), 257
testing of (TIO), 258
Write Direct (WRD), 259

JCL, ASSGN statement (in DOS), 560
basic intent of, 8
coding rules for, 550
compiler specifications in, 553
data control block (DCB) description, 558
DD description, 558
device names in, 561
EXEC command, 552, 554
GO command, 557
job identification for, 550
job specification, 551
OPTION specification (for DOS), 556
organization of jobs under, 560
types of statements in, 39
Job, automatic handling of, 4
defined, 31
scheduling of, 32
steps in a, 31
Job Control Language (*See* JCL)
Job scheduler, 32
Job step, 31

Large Core Storage (LCS), 19
Linkage editor, 38
List-directed input/output, in PL/I, 487
Literal values, in ALGOL, 532
in Assembler Language, 105
in COBOL, 362
in FORTRAN, 322
in PL/I, 457
Location counter, 107
Logical operations, in ALGOL, 533
in Assembler Language, 226
Logical operations (*cont.*)
in FORTRAN, 349
in PL/I, 474
Logical records vs. physical records, 34
Loops, in ALGOL, 540
in Assembler Language, 161
in COBOL, 410
in FORTRAN, 304
in PL I, 480

Machine check interrupts, 82
Machine check mask, 85
Machine instructions, table of, 71
Machine states, 82
Macro instructions, CLOSE, 282
DTFDA, 279
DTFSR, 275
GET, 284
OPEN, 281
PRTOV, 286
PUT, 284
PUTX, 286
READ, 283
RELSE, 286
TRUNC, 286
WRITE, 283
Magnetic disk, organizational properties of, in COBOL, 400, 447
Magnetic tape, and COBOL, 379
manipulation of, in FORTRAN, 327
organization of data for, 279
Magnetic-tape units, operating characteristics of, 25
Mark-sense readers, 27
Matrices (*See* arrays)
Memory (*See* storage)
Miniaturization of System/360 circuitry, 3
Mnemonic codes, organization of, in Assembler Language, 98
Multiplexer channels, 11
Multiprocessing, 6
Multiprogramming, 6

Numbers, binary, 57
complement notation for, 58
complex, 346

Numbers (*cont.*)
conversion from zoned- to packed-decimal form, 174
double precision, in FORTRAN, 291
floating-point, 60, 207
integers, in FORTRAN, 291
internal representation of, 53
negative, 57
packed-decimal, 55, 171
real, in FORTRAN, 291
rounding of, 189
zoned-decimal, 54

Operating system (*See* also Software)
privileged instructions for, 33
scheduling of jobs in, 32
supervision of jobs in, 32
use of a catalog in, 34
utility programs provided by, 37
Organization of System/360, 1
Overlapped operation, 5

Packed decimals, conversion from zoned-decimal formats, 174
conversion to zoned-decimal formats, 175
format for, 55
representation of, 171
rounding of, 189
scaling of, for division, 183, 191
specification of, 170
PL/I, ALIGNED attribute (for storage allocation), 529
ALLOCATE statement, 527
arguments in subprograms, 515
arithmetic expressions in, 462
array cross sections, 467
array subscripts, 466, 468
assignment statements, 462
AUTOMATIC attribute (for storage allocation), 525
BACKWARDS attribute (for files), 500
base, 458
BASED variables, 506
BEGIN block, 521
BINARY attribute, 459
BIT attribute, 459

PL/I (*cont.*)
buffered and unbuffered files, 504
BUFFERED attribute (for files), 500
built-in functions, 463
built-in functions (for bit handling), 475
built-in functions (for character handling), 472
CALL statement, 517
CHARACTER attribute, 460
character set for, 455
CLOSE statement, 502
coding rules for, 454
comments, 457
concatenation of character strings in, 471
conditional expressions, 477
constants, 457
CONTROLLED attribute (for file handling), 506
cyclic operations in, 480
data-directed input/output, 489
debugging aids in, 503
decimal attribute, 459
declaration of arrays, 465
declaration of bit strings, 459
declaration of structures, 470
DECLARE statement, 459, 462
defaults for declarations, 462
DEFINED attribute (for storage allocation), 528
DELETE FILE statement, 508
description of input in, 491
description of output in, 493
DIRECT attribute (for files), 499
direct files, 508
DISPLAY statement, 510
DO group, 479
DO loop, 480
DO loops in GET and PUT statements, 498
DO statement, 479
dynamic storage allocation in, 524
edit-directed input/output, 490
editing of output, 488, 493
ELSE statement, 478
enabling and disabling of interrupts in, 522

PL/I (*cont.*)
END statement, 457, 480, 516
ENTRY statement, 516
EXCLUSIVE attribute (for files), 501
EXTERNAL attribute, 520
file descriptions in, 499
FIXED attribute, 459
FLOAT attribute, 460
FORMAT statement, 496
FREE statement, 527
FROM keyword (with WRITE), 504
function procedures, 514
generic functions, 464
GET DATA statement, 489
GET EDIT statement, 491
GET FILE statement, 487
GET LIST statement, 487
GET STRING statement, 497
GO TO statement, 476
IDENT specification (for files), 502
IF statement, 476, 479
IGNORE keyword (with READ), 504
INITIAL attribute, 460
INPUT attribute (for files), 500
input/output of arrays and structures, 497
interrupt handling in, 522
INTO keyword (with READ), 504
KEY specification (with DELETE), 508
KEYED attribute (for files), 499
KEYFROM keyword (with WRITE), 505
KEYTO keyword (with READ), 504
label arrays, 497
LABEL attribute, 476
LIKE attribute (for storage allocation), 529
LINE specification, in list I/O, 488
LINESIZE specification (with PRINT files), 502
list-directed input/output, 487
LOCATE statement, 506
logical operations in, 474
loops in, 480
manipulation of arrays, 466
mode, 458
nested DO loops, 483
ON statement, 503

PL/I (*cont.*)
OPEN statement, 502
operational symbols, 455
organization of programs in, 457, 514, 518
organization of structures in, 469
OUTPUT attribute (for files), 500
PACKED attribute (for storage allocation), 529
PAGE specification, in list I/O, 488
PAGESIZE specification (with PRINT files), 502
parameters in subprograms, 515
PICTURE attribute, 461
PICTURE specification (for describing output), 494
POINTER attribute, 527
pointers in, 506
POSITION attribute (for storage allocation), 528
precision, 458
PRINT attribute (for files), 500
PROCEDURE statement, 457
PUT DATA statement, 489
PUT EDIT statement, 493
PUT FILE statement, 487
PUT LIST statement, 487
READ statement, 503
REAL attribute, 459
RECORD attribute (for files), 499
record-oriented input/output, 486
relation among procedures, 518
relational operators, 477
REPLY statement (with DISPLAY), 510
representation of characters in, 459
representation of English currency values in, 458
representation of numbers in, 458
RETURN statement, 515
scale, 458
scope of procedures in, 520
SEQUENTIAL attribute (for files), 499
SET keyword (with READ), 506
SIGNAL statement, 522
SKIP specification, in list I/O, 488
statement labels, 456, 476
STATIC attribute (for storage allocation), 525

PL/I (*cont.*)
storage allocation in, 459, 465, 470
STREAM attribute (for files), 499
stream-oriented input/output, 486
subroutines in, 517
UNBUFFERED attribute (for files), 500
UPDATE attribute (for files), 500, 507
VARYING attribute (for character strings), 461
WRITE statement, 503
Pointers, in ALGOL, 547
in PL/I, 506, 527
Printers, operating characteristics of, 22
Privileged instruction, 33
Problem state indicator, 86
Program compatibility in System/360 models, 10
Program interrupts, 81
Program mask, 86
Program Status Word (*See* PSW)
Programs, organization of, in ALGOL, 541
organization of, in Assembler Language, 127, 271
organization of, in COBOL, 361
organization of, in FORTRAN, 345
organization of, in PL/I, 457, 514
Protection key, 85
PSW, current, 84
format of, 84
"old," 87
"new," 87

Random number generating routine in Assembler Language, 237
Records, handling of, in COBOL, 376
in JCL, 558
organization of, in PL/I, 503
Registers, 51
Relational operators, in ALGOL, 533
in FORTRAN, 351
in PL/I, 477
Relocatability, 38, 99
Report writing, in COBOL, 431
Rounding, 189, 367
RR instruction format, 65
RS instruction format, 67
RX instruction format, 66

Scaling of packed-decimal values for division, 183, 191
Searching techniques in COBOL, EXAMINE, 382
Selector channels, 11
Sequential operators, in ALGOL, 533
SI instruction format, 68
Software (*See* also Operating system)
basic organization of, 31
data management by, 34
major components of, in System/360, 32
task management by, 33
Sorting, in COBOL, 426
Source program library, in COBOL, 449
Specificators, in ALGOL, 533
SS instruction format, 69
Standardization of System/360 instruction sets, 10
Statement functions, 332
Storage, 47
Storage allocation, in ALGOL, 540
in Assembler Language, 112, 122
in FORTRAN, 292
in PL/I, 459, 465, 470
for COBOL arrays, 409
Storage protection feature, 7, 20
Structures, in PL/I, 469
Subprograms, in COBOL, 417
in FORTRAN, 332
in PL/I, 514
Subroutines, in Assembler Language, 155
in COBOL, 417
in FORTRAN, 338
in PL/I, 517
Supervisor, 32
Supervisor Call interrupts, 81
SYSIPT (SYSIN), 261
SYSLST, 261
SYSOPT (SYSOUT), 261
SYSRDR, 261
System mask bits, 85
System/360, access time in, 19

System/360 (*cont.*)
addressing methods in, 49
as a fixed word length machine, 56
as a variable word length machine, 54
base-displacement addressing in, 50
basic branching mechanism in, 148
basic instruction formats in, 64
boundary alignment in, 117
card readers and punches for, 21
cathode ray devices for, 21
classes of interrupts in, 81
compatibility between models, 8
complementation in, 58
CPU organization in, 15
CPU sizes in, 16
data cells for, 37
direct-access devices for, 27, 30
effect of miniaturization on, 3
general organization of, 1
general purpose registers in, 50
index registers in, 164
interrupt handling in, 7
internal code for characters in, 45
large core storage (LCS) in, 19
machine instructions in, 71
machine states in, 82
magnetic disk drives for, 27
magnetic drums for, 27
magnetic tape devices for, 24
models included in, 2
nonstop operation in, 4
System/360 (*cont.*)
organization of storage in, 47
organizational advantages of, 2
overlapped operation in, 5
printers for, 21
regulation of I/O activities in, 89
relocatability in, 38
software organization in, 32
speed of, 19
storage protection feature in, 7, 20
task management in, 33
universality of, 11
use of registers in, 50

Task management, 33

Utility programs, 37

Variables, in ALGOL, 533
declaration of, in PL/I, 458
initialization of, in PL/I, 460
names for, in PL/I, 462
naming of, in COBOL, 423
naming of, in FORTRAN, 347

Wait state indicator, 86

Zoned decimals, conversion from packed-decimal form, 175
conversion to packed-decimal form, 174
format for, 54